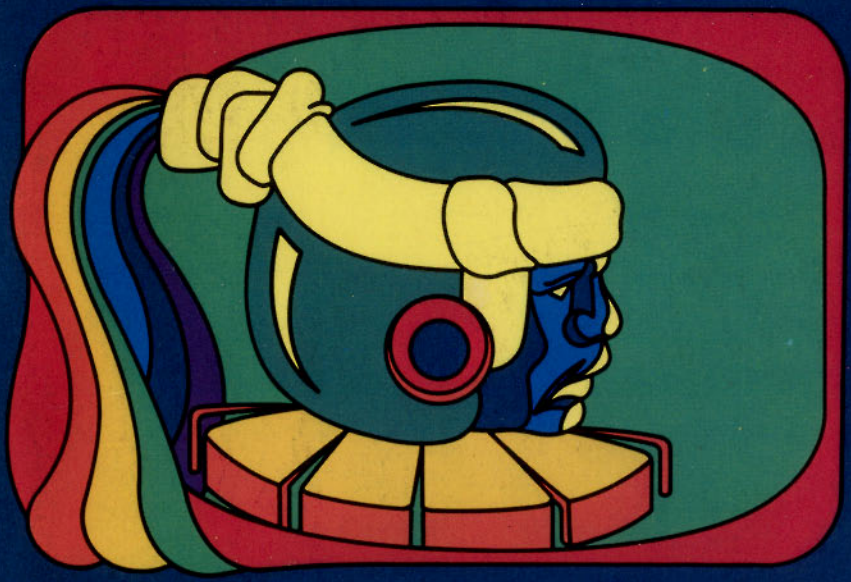
OSBORNE & ASSOCIATES

31003

# 8080A/8085

## ASSEMBLY LANGUAGE PROGRAMMING



Lance A. Leventhal

8080A/8085 ASSEMBLY LANGUAGE PROGRAMMING

31003

# 8080A/8085
# ASSEMBLY LANGUAGE PROGRAMMING

Lance A. Leventhal

Osborne & Associates, Inc.
Berkeley, California

Copyright © 1978 by Adam Osborne & Associates, Incorporated.

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in any retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publishers.

Published by Adam Osborne & Associates, Incorporated
P.O. Box 2036, Berkeley, California 94702

Portions of Chapter 3 have been reprinted from Adam Osborne & Associates, Incorporated, 8080 Programming for Logic Design with the permission and consent of Adam Osborne & Associates, Incorporated.

## DISTRIBUTORS OF OSBORNE & ASSOCIATES INC. PUBLICATIONS

**For information on translations and book distributors outside of the United States of America, please write:**

**Osborne & Associates, Inc.**
P.O. Box 2036
Berkeley, California 94702
United States of America
(415) 548-2805

DISTRIBUTORS

L. P. ENTERPRISES
313 [illegible] ROAD
ILFORD, Essex, IG1 1PJ
Tel: 01-555 1001

**Production Staff**
**Osborne & Associates, Inc.**

| | |
|---|---|
| Curt Ingraham | Technical Editor |
| Vicki Mitchell<br>Penny Lawrence<br>Jerry Beach | Typography |
| Karen de Robinson | Graphics |
| Laurie Battle | Proofreader |
| Mary Borchers | Coordinator |

Cover Design by K.L.T. van Genderen

Printed by Rotary Offset Printers
San Francisco, California

# ACKNOWLEDGMENTS

The author would like to acknowledge the following people:

Mr. William Tester of Grossmont College, who is responsible for the support which made this book possible; Mr. Colin Walsh of Grossmont College, who helped throughout with the design and development of the programming examples; Mr. Curt Ingraham of Osborne & Associates, who made many editorial corrections; Mrs. Teddy Ferguson, who typed the original problem assignments; Mr. Stanley Rogers of the Society for Computer Simulation, who quietly but convincingly suggested many improvements in the author's writing style; and his wife Donna, for her patience and understanding throughout the writing of this book.

Others who provided assistance and suggestions were Mr. Jeffrey Haight, Prof. Nicholas Panos, Mr. David Bulman, Mrs. Kati Bulman, Mr. Bernard Laffreniere, Mr. Charles Robe, Mr. Michael Viehman, Mr. Richard Evans, Mr. Frederick Lepow and Mr. William Long. Other students and colleagues also helped to keep the author on the right track.

The author, of course, bears responsibility for any remaining errors, misconceptions and misinterpretations.

This book is dedicated to the memory of Bayar Goodman, who would have appreciated and enjoyed this new technology.

# TABLE OF CONTENTS


| CHAPTER | | PAGE |
|---|---|---|
| 1 | INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING | 1-1 |
| | HOW THIS BOOK HAS BEEN PRINTED | 1-1 |
| | THE MEANING OF INSTRUCTIONS | 1-1 |
| | A COMPUTER PROGRAM | 1-2 |
| | THE PROGRAMMING PROBLEM | 1-2 |
| | USING OCTAL OR HEXADECIMAL | 1-3 |
| | INSTRUCTION CODE MNEMONICS | 1-5 |
| | THE ASSEMBLER PROGRAM | 1-5 |
| | ADDITIONAL FEATURES OF ASSEMBLERS | 1-6 |
| | DISADVANTAGES OF ASSEMBLY LANGUAGE | 1-7 |
| | HIGH-LEVEL LANGUAGES | 1-7 |
| | ADVANTAGES OF HIGH-LEVEL LANGUAGES | 1-8 |
| | DISADVANTAGES OF HIGH-LEVEL LANGUAGES | 1-8 |
| | HIGH-LEVEL LANGUAGES FOR MICROPROCESSORS | 1-10 |
| | WHICH LEVEL SHOULD YOU USE? | 1-11 |
| | HOW ABOUT THE FUTURE? | 1-12 |
| | WHY THIS BOOK? | 1-12 |
| 2 | ASSEMBLERS | 2-1 |
| | FEATURES OF ASSEMBLERS | 2-1 |
| | ASSEMBLER INSTRUCTIONS | 2-1 |
| | LABELS | 2-2 |
| | ASSEMBLER OPERATION CODES (MNEMONICS) | 2-4 |
| | PSEUDO-OPERATIONS | 2-4 |
| | THE DATA PSEUDO-OPERATION | 2-5 |
| | THE EQUATE (OR EQUALS) PSEUDO-OPERATION | 2-6 |
| | THE ORIGIN PSEUDO-OPERATION | 2-7 |
| | THE RESERVE PSEUDO-OPERATION | 2-7 |
| | HOUSEKEEPING PSEUDO-OPERATIONS | 2-8 |
| | LABELS WITH PSEUDO-OPERATIONS | 2-9 |
| | ADDRESSES AND THE OPERAND FIELD | 2-9 |
| | CONDITIONAL ASSEMBLY | 2-11 |
| | MACROS | 2-11 |
| | COMMENTS | 2-13 |
| | TYPES OF ASSEMBLERS | 2-14 |
| | ERRORS | 2-15 |
| | LOADERS | 2-15 |
| 3 | THE 8080A AND 8085 ASSEMBLY LANGUAGE | 3-1 |
| | INSTRUCTION SETS | 3-1 |
| | CPU REGISTERS AND STATUS FLAGS | 3-2 |
| | 8080A AND 8085 MEMORY ADDRESSING | 3-3 |
| | ABBREVIATIONS | 3-8 |
| | STATUS | 3-9 |
| | INSTRUCTION MNEMONICS | 3-9 |
| | INSTRUCTION OBJECT CODES | 3-9 |
| | INSTRUCTION EXECUTION TIMES AND CODES | 3-9 |
| | ACI — ADD WITH CARRY IMMEDIATE TO ACCUMULATOR | 3-19 |
| | ADC — ADD REGISTER OR MEMORY WITH CARRY TO ACCUMULATOR | 3-20 |
| | ADD — ADD REGISTER OR MEMORY TO ACCUMULATOR | 3-22 |

# TABLE OF CONTENTS (Continued)


| CHAPTER | | PAGE |
|---|---|---|
| 3 (Cont.) | ADI — ADD IMMEDIATE TO ACCUMULATOR | 3-24 |
| | ANA — AND REGISTER OR MEMORY WITH ACCUMULATOR | 3-25 |
| | ANI — AND IMMEDIATE WITH ACCUMULATOR | 3-27 |
| | CALL — CALL THE SUBROUTINE IDENTIFIED IN OPERAND | 3-38 |
| | CC — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE CARRY STATUS EQUALS 1 | 3-29 |
| | CM — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE SIGN STATUS EQUALS 1 | 3-29 |
| | CMA — COMPLEMENT THE ACCUMULATOR | 3-30 |
| | CMC — COMPLEMENT THE CARRY STATUS | 3-31 |
| | CMP — COMPARE REGISTER OR MEMORY WITH ACCUMULATOR | 3-32 |
| | CND — CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND, BUT ONLY IF THE CARRY STATUS EQUALS 0 | 3-33 |
| | CNZ — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE ZERO STATUS EQUALS 0 | 3-34 |
| | CP — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE SIGN STATUS EQUALS 0 | 3-34 |
| | CPE — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE PARITY STATUS EQUALS 1 | 3-35 |
| | CPI — COMPARE ACCUMULATOR CONTENTS WITH IMMEDIATE DATA | 3-36 |
| | CPO — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE PARITY STATUS EQUALS 0 | 3-37 |
| | CZ — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE ZERO STATUS EQUALS 1 | 3-37 |
| | DAA — DECIMAL ADJUST ACCUMULATOR | 3-38 |
| | DAD — ADD A REGISTER PAIR TO H AND L | 3-39 |
| | DCR — DECREMENT REGISTER OR MEMORY CONTENTS | 3-40 |
| | DCX — DECREMENT REGISTER PAIR | 3-42 |
| | DI — DISABLE INTERRUPTS | 3-43 |
| | EI — ENABLE INTERRUPTS | 3-43 |
| | HLT — HALT | 3-45 |
| | IN — INPUT TO ACCUMULATOR | 3-46 |
| | INR — INCREMENT REGISTER OR MEMORY CONTENTS | 3-47 |
| | INX — INCREMENT REGISTER PAIR | 3-49 |
| | JC — JUMP IF CARRY | 3-50 |
| | JM — JUMP IF MINUS | 3-50 |
| | JMP — JUMP TO THE INSTRUCTION IDENTIFIED IN THE OPERAND | 3-51 |
| | JNC — JUMP IF NO CARRY | 3-51 |
| | JNZ — JUMP IF NOT ZERO | 3-52 |
| | JPE — JUMP IF PARITY EVEN | 3-53 |

# TABLE OF CONTENTS (Continued)


| CHAPTER | | PAGE |
|---|---|---|
| 3 (Cont.) | JPO — JUMP IF PARITY ODD | 3-53 |
| | JZ — JUMP IF ZERO | 3-54 |
| | LDA — LOAD ACCUMULATOR FROM MEMORY USING DIRECT ADDRESSING | 3-55 |
| | LDAX — LOAD ACCUMULATOR FROM MEMORY LOCATION ADDRESSED BY REGISTER PAIR | 3-56 |
| | LHLD — LOAD H AND L REGISTERS DIRECT | 3-57 |
| | LXI — LOAD A 16-BIT VALUE, IMMEDIATE, INTO A REGISTER PAIR | 3-58 |
| | LXI — MOVE DATA | 3-59 |
| | MVI — LOAD DATA IMMEDIATE INTO REGISTER OR MEMORY | 3-61 |
| | NOP — NONO OPERATION | 3-63 |
| | ORA — OR REGISTER OR MEMORY WITH ACCUMULATOR | 3-64 |
| | ORI — OR IMMEDIATE WITH ACCUMULATOR | 3-66 |
| | OUT — OUTPUT FROM ACCUMULATOR | 3-67 |
| | PCHL — JUMP TO ADDRESS SPECIFIED BY HL | 3-68 |
| | POP — READ FROM THE TOP OF THE STACK | 3-69 |
| | PUSH — WRITE TO THE TOP OF THE STACK | 3-70 |
| | RAL — ROTATE ACCUMULATOR LEFT THROUGH CARRY | 3-71 |
| | RAR — ROTATE ACCUMULATOR RIGHT THROUGH CARRY | 3-72 |
| | RC — RETURN IF THE CARRY STATUS EQUALS 1 | 3-73 |
| | RIM — READ INTERRUPT MASK | 3-74 |
| | RLC — ROTATE ACCUMULATOR LEFT | 3-75 |
| | RM — RETURN IF THE SIGN STATUS EQUALS 1 | 3-76 |
| | RNC — RETURN IF THE CARRY STATUS EQUALS 0 | 3-76 |
| | RNZ — RETURN IF THE ZERO STATUS EQUALS 0 | 3-77 |
| | RP — RETURN IF THE SIGN STATUS EQUALS 0 | 3-77 |
| | RPE — RETURN IF THE PARITY STATUS EQUALS 1 | 3-78 |
| | RPO — RETURN IF THE PARITY STATUS EQUALS 0 | 3-78 |
| | RRC — ROTATE ACCUMULATOR RIGHT | 3-79 |
| | RST — RESTART | 3-80 |
| | RZ — RETURN IF THE ZERO STATUS EQUALS 1 | 3-81 |
| | SBB — SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR WITH BORROW | 3-82 |
| | SBI — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR WITH BORROW | 3-84 |
| | SHLD — STORE H AND L REGISTERS DIRECT | 3-85 |
| | SIM — SET INTERRUPT MASK | 3-86 |
| | SPHL — LOAD THE STACK POINTER FROM THE H AND L REGISTERS | 3-87 |
| | STA — STORE ACCUMULATOR IN MEMORY USING DIRECT ADDRESSING | 3-88 |
| | STAX — STORE ACCUMULATOR IN THE MEMORY LOCATION ADDRESSED BY A REGISTER PAIR | 3-89 |
| | STC — SET CARRY STATUS | 3-90 |
| | SUB — SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR | 3-91 |
| | SUI — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR | 3-93 |

# TABLE OF CONTENTS (Continued)


| CHAPTER | | PAGE |
|---|---|---|
| 3 (Cont.) | XCHG — EXCHANGE DE AND HL REGISTERS' CONTENTS | 3-94 |
| | XRA — EXCLUSIVE-OR REGISTER OR MEMORY WITH ACCUMULATOR | 3-95 |
| | XRI — EXCLUSIVE-OR IMMEDIATE DATA WITH ACCUMULATOR | 3-97 |
| | XTHL — EXCHANGE TOP OF STACK WITH HL | 3-98 |
| | INTEL 8080A AND 8085 ASSEMBLER CONVENTIONS | 3-99 |
| | ASSEMBLER FIELD STRUCTURE | 3-99 |
| | LABELS | 3-99 |
| | PSEUDO-OPERATIONS | 3-99 |
| | LABELS WITH PSEUDO-OPERATIONS | 3-101 |
| | ADDRESSES | 3-101 |
| | CONDITIONAL ASSEMBLY | 3-102 |
| | MACROS | 3-103 |
| | BNPF FORMAT | 3-103 |
| 4 | SIMPLE PROGRAMS | 4-1 |
| | GENERAL FORMAT OF EXAMPLES | 4-1 |
| | GUIDELINES FOR PROBLEMS | 4-2 |
| | EXAMPLES | 4-3 |
| | ONES COMPLEMENT | 4-3 |
| | 8-BIT ADDITION | 4-4 |
| | SHIFT LEFT ONE BIT | 4-5 |
| | MASK OUT LEAST SIGNIFICANT FOUR BITS | 4-6 |
| | CLEAR A MEMORY LOCATION | 4-7 |
| | WORD DISASSEMBLY | 4-7 |
| | FIND LARGER OF TWO NUMBERS | 4-9 |
| | 16-BIT ADDITION | 4-10 |
| | TABLE OF SQUARES | 4-12 |
| | 16-BIT ONES COMPLEMENT | 4-14 |
| | PROBLEMS | 4-15 |
| | TWOS COMPLEMENT | 4-15 |
| | 8-BIT SUBTRACTION | 4-15 |
| | SHIFT LEFT TWO BITS | 4-15 |
| | MASK OUT MOST SIGNIFICANT FOUR BITS | 4-15 |
| | SET A MEMORY LOCATION TO ALL ONES | 4-15 |
| | WORD ASSEMBLY | 4-15 |
| | FIND SMALLER OF TWO NUMBERS | 4-16 |
| | 24-BIT ADDITION | 4-16 |
| | SUM OF SQUARES | 4-16 |
| | 16-BIT TWOS COMPLEMENT | 4-17 |

# TABLE OF CONTENTS (Continued)


| CHAPTER | | PAGE |
|---|---|---|
| 5 | SIMPLE PROGRAM LOOPS | 5-1 |
| | EXAMPLES | 5-4 |
| | SUM OF DATA | 5-4 |
| | 16-BIT SUM OF DATA | 5-7 |
| | NUMBER OF NEGATIVE ELEMENTS | 5-9 |
| | FIND MAXIMUM | 5-11 |
| | JUSTIFY A BINARY FRACTION | 5-14 |
| | PROBLEMS | 5-16 |
| | CHECKSUM OF DATA | 5-16 |
| | SUM OF 16-BIT DATA | 5-16 |
| | NUMBER OF ZERO, POSITIVE AND NEGATIVE NUMBERS | 5-17 |
| | FIND MINIMUM | 5-17 |
| | COUNT 1 BITS | 5-17 |
| 6 | CHARACTER-CODED DATA | 6-1 |
| | EXAMPLES | 6-3 |
| | LENGTH OF A STRING OF CHARACTERS | 6-3 |
| | FIND FIRST NON-BLANK CHARACTER | 6-7 |
| | REPLACE LEADING ZEROS WITH BLANKS | 6-11 |
| | ADD EVEN PARITY TO ASCII CHARACTERS | 6-13 |
| | PATTERN MATCH | 6-16 |
| | PROBLEMS | 6-19 |
| | LENGTH OF A TELETYPEWRITER MESSAGE | 6-19 |
| | FIND LAST NON-BLANK CHARACTER | 6-19 |
| | TRUNCATE DECIMAL STRING TO INTEGER FORM | 6-20 |
| | CHECK EVEN PARITY IN ASCII CHARACTERS | 6-20 |
| | STRING COMPARISON | 6-21 |
| 7 | CODE CONVERSION | 7-1 |
| | EXAMPLES | 7-1 |
| | HEX TO ASCII | 7-1 |
| | DECIMAL TO 7-SEGMENT | 7-4 |
| | ASCII TO DECIMAL | 7-7 |
| | BCD TO BINARY | 7-8 |
| | ASCII STRING TO BINARY NUMBER | 7-9 |
| | PROBLEMS | 7-13 |
| | ASCII TO HEX | 7-13 |
| | 7-SEGMENT TO DECIMAL | 7-13 |
| | DECIMAL TO ASCII | 7-13 |
| | BINARY TO BCD | 7-13 |
| | BINARY NUMBER TO ASCII STRING | 7-14 |

# TABLE OF CONTENTS (Continued)


| CHAPTER | | PAGE |
|---|---|---|
| 8 | ARITHMETIC PROBLEMS | 8-1 |
| | EXAMPLES | 8-2 |
| | MULTIPLE-PRECISION ADDITION | 8-2 |
| | DECIMAL ADDITION | 8-4 |
| | 8-BIT BINARY MULTIPLICATION | 8-6 |
| | 8-BIT BINARY DIVISION | 8-11 |
| | SELF-CHECKING NUMBERS — DOUBLE ADD DOUBLE, MOD 10 | 8-16 |
| | PROBLEMS | 8-20 |
| | MULTIPLE-PRECISION SUBTRACTION | 8-20 |
| | DECIMAL SUBTRACTION | 8-20 |
| | 8-BIT BY 16-BIT BINARY MULTIPLICATION | 8-21 |
| | SIGNED BINARY DIVISION | 8-21 |
| | SELF-CHECKING NUMBERS ALIGNED 1, 3, 7 MOD 10 | 8-22 |
| 9 | TABLES AND LISTS | 9-1 |
| | EXAMPLES | 9-1 |
| | ADD ENTRY TO LIST | 9-1 |
| | CHECK AN ORDERED LIST | 9-4 |
| | REPLACING A CHAIN WITH DATA | 9-7 |
| | 8-BIT SORT | 9-10 |
| | USING A JUMP TABLE WITH A KEY | 9-14 |
| | PROBLEMS | 9-16 |
| | REMOVE ENTRY FROM LIST | 9-16 |
| | ADD ENTRY TO ORDERED LIST | 9-17 |
| | ADD ELEMENT TO CHAINED LIST | 9-17 |
| | 16-BIT SORT | 9-18 |
| | USING AN ORDERED JUMP TABLE | 9-18 |
| 10 | SUBROUTINES | 10-1 |
| | SUBROUTINE DOCUMENTATION | 10-2 |
| | EXAMPLES | 10-3 |
| | HEX TO ASCII | 10-3 |
| | LENGTH OF A STRING OF CHARACTERS | 10-7 |
| | ADD EVEN PARITY TO ASCII CHARACTERS | 10-10 |
| | PATTERN MATCH | 10-14 |
| | MULTIPLE-PRECISION ADDITION | 10-18 |
| | PROBLEMS | 10-21 |
| | ASCII TO HEX | 10-21 |
| | LENGTH OF A TELETYPE MESSAGE | 10-21 |
| | CHECK EVEN PARITY IN ASCII CHARACTERS | 10-21 |
| | STRING COMPARISON | 10-22 |
| | DECIMAL SUBTRACTION | 10-23 |

# TABLE OF CONTENTS (Continued)


| CHAPTER | | PAGE |
|---|---|---|
| 11 | INPUT/OUTPUT | 11-1 |
| | TIMING INTERVALS (DELAYS) | 11-8 |
| | DELAY ROUTINES | 11-8 |
| | EXAMPLES | 11-9 |
| | DELAY PROGRAM | 11-9 |
| | A PUSHBUTTON (OR SPST MOMENTARY SWITCH) | 11-11 |
| | A TOGGLE (SPDT) SWITCH | 11-17 |
| | A MULTIPLE-POSITION (ROTARY, SELECTOR, OR THUMBWHEEL) SWITCH | 11-21 |
| | A SINGLE LED | 11-27 |
| | 7-SEGMENT LED DISPLAY | 11-30 |
| | PROBLEMS | 11-36 |
| | AN ON-OFF PUSHBUTTON | 11-36 |
| | DEBOUNCING A SWITCH IN SOFTWARE | 11-36 |
| | CONTROL FOR A ROTARY SWITCH | 11-37 |
| | RECORD SWITCH POSITIONS ON LIGHTS | 11-37 |
| | COUNT ON A 7-SEGMENT DISPLAY | 11-37 |
| | MORE COMPLEX I/O DEVICES | 11-38 |
| | EXAMPLES | 11-40 |
| | AN UNENCODED KEYBOARD | 11-40 |
| | AN ENCODED KEYBOARD | 11-48 |
| | A DIGITAL-TO-ANALOG CONVERTER | 11-52 |
| | AN ANALOG-TO-DIGITAL CONVERTER | 11-56 |
| | A TELETYPEWRITER (TTY) | 11-60 |
| | PROBLEMS | 11-69 |
| | SEPARATING CLOSURES FROM AN UNENCODED KEYBOARD | 11-69 |
| | READ A SENTENCE FROM AN ENCODED KEYBOARD | 11-69 |
| | A VARIABLE AMPLITUDE SQUARE WAVE GENERATOR | 11-69 |
| | AVERAGING ANALOG READINGS | 11-70 |
| | A 30 CHARACTERS-PER-SECOND TERMINAL | 11-70 |
| 12 | INTERRUPTS | 12-1 |
| | 8080 INTERRUPT SYSTEM | 12-3 |
| | THE RESTART (RST) INSTRUCTION | 12-3 |
| | 8085 INTERRUPT SYSTEM | 12-6 |
| | 8214 PRIORITY INTERRUPT CONTROL UNIT | 12-6 |
| | 8259 PROGRAMMABLE INTERRUPT CONTROLLER | 12-9 |
| | EXAMPLES | 12-11 |
| | A STARTUP INTERRUPT | 12-11 |
| | A KEYBOARD INTERRUPT | 12-14 |
| | A PRINTER INTERRUPT | 12-17 |
| | A REAL-TIME CLOCK INTERRUPT | 12-19 |
| | A TELETYPEWRITER INTERRUPT | 12-23 |
| | MORE GENERAL SERVICE ROUTINES | 12-24 |
| | PROBLEMS | 12-27 |
| | A TEST INTERRUPT | 12-27 |
| | A KEYBOARD INTERRUPT | 12-27 |
| | A PRINTER INTERRUPT | 12-27 |
| | A REAL-TIME CLOCK INTERRUPT | 12-27 |
| | A TELETYPEWRITER INTERRUPT | |

# TABLE OF CONTENTS (Continued)


| CHAPTER | | PAGE |
|---|---|---|
| 13 | PROBLEM DEFINITION AND PROGRAM DESIGN | 13-1 |
| | THE TASKS OF SOFTWARE DEVELOPMENT | 13-1 |
| | DEFINITION OF THE STAGES | 13-3 |
| | PROBLEM DEFINITION | 13-3 |
| | DEFINING THE INPUTS | 13-3 |
| | DEFINING THE OUTPUTS | 13-4 |
| | PROCESSING SECTION | 13-4 |
| | ERROR HANDLING | 13-5 |
| | HUMAN FACTORS | 13-5 |
| | EXAMPLES | 13-6 |
| | RESPONSE TO A SWITCH | 13-6 |
| | A SWITCH-BASED MEMORY LOADER | 13-7 |
| | A VERIFICATION TERMINAL | 13-10 |
| | REVIEW OF PROBLEM DEFINITION | 13-13 |
| | PROGRAM DESIGN | 13-14 |
| | FLOWCHARTING | 13-14 |
| | EXAMPLES | 13-16 |
| | RESPONSE TO A SWITCH | 13-16 |
| | THE SWITCH-BASED MEMORY LOADER | 13-17 |
| | THE CREDIT-VERIFICATION TERMINAL | 13-19 |
| | MODULAR PROGRAMS | 13-23 |
| | EXAMPLES | 13-24 |
| | RESPONSE TO A SWITCH | 13-24 |
| | THE SWITCH-BASED MEMORY LOADER | 13-24 |
| | THE VERIFICATION TERMINAL | 13-24 |
| | REVIEW OF MODULAR PROGRAMMING | 13-25 |
| | STRUCTURED PROGRAMMING | 13-25 |
| | EXAMPLES | 13-29 |
| | RESPONSE TO A SWITCH | 13-29 |
| | THE SWITCH-BASED MEMORY LOADER | 13-30 |
| | THE CREDIT-VERIFICATION TERMINAL | 13-31 |
| | REVIEW OF STRUCTURED PROGRAMMING | 13-34 |
| | TOP-DOWN DESIGN | 13-35 |
| | EXAMPLES | 13-36 |
| | RESPONSE TO A SWITCH | 13-36 |
| | THE SWITCH-BASED MEMORY LOADER | 13-37 |
| | THE TRANSACTION TERMINAL | 13-38 |
| | REVIEW OF TOP-DOWN DESIGN | 13-39 |
| | REVIEW OF PROBLEM DEFINITION AND PROGRAM DESIGN | 13-40 |
| | REFERENCES | 13-41 |

# TABLE OF CONTENTS (Continued)


| CHAPTER | | PAGE |
|---|---|---|
| 14 | DEBUGGING AND TESTING | 14-1 |
| | SIMPLE DEBUGGING TOOLS | 14-1 |
| | MORE ADVANCED DEBUGGING TOOLS | 14-6 |
| | DEBUGGING WITH CHECKLISTS | 14-8 |
| | LOOKING FOR ERRORS | 14-9 |
| | DEBUGGING EXAMPLES | 14-12 |
| | DECIMAL TO 7-SEGMENT CONVERSION | 14-12 |
| | SORT INTO DECREASING ORDER | 14-15 |
| | INTRODUCTION TO TESTING | 14-21 |
| | TOOLS FOR TESTING | 14-22 |
| | SELECTING TEST DATA | 14-22 |
| | TESTING EXAMPLES | 14-23 |
| | SORT PROGRAM | 14-23 |
| | SELF-CHECKING NUMBERS | 14-24 |
| | TESTING PRECAUTIONS | 14-24 |
| | CONCLUSIONS | 14-24 |
| | REFERENCES | 14-25 |
| 15 | DOCUMENTATION AND RE-DESIGN | 15-1 |
| | SELF-DOCUMENTING PROGRAMS | 15-1 |
| | COMMENTS | 15-2 |
| | COMMENTING EXAMPLES | 15-4 |
| | MULTIPLE-PRECISION ADDITION | 15-4 |
| | TELETYPEWRITER OUTPUT | 15-5 |
| | FLOWCHARTS AS DOCUMENTATION | 15-6 |
| | STRUCTURED PROGRAMS AS DOCUMENTATION | 15-6 |
| | MEMORY MAPS | 15-7 |
| | PARAMETER AND DEFINITION LISTS | 15-7 |
| | LIBRARY ROUTINES | 15-8 |
| | LIBRARY EXAMPLES | 15-9 |
| | SUM OF DATA | 15-9 |
| | DECIMAL TO 7-SEGMENT CONVERSION | 15-10 |
| | DECIMAL SUM | 15-11 |
| | TOTAL DOCUMENTATION | 15-12 |
| | RE-DESIGN | 15-12 |
| | REORGANIZING TO USE LESS MEMORY | 15-13 |
| | REORGANIZING TO USE LESS TIME | 15-13 |
| | MAJOR REORGANIZATIONS | 15-14 |
| 16 | SAMPLE PROJECTS | 16-1 |
| | PROJECT #1: A DIGITAL STOPWATCH | 16-1 |
| | PROJECT #2: A DIGITAL THERMOMETER | 16-15 |

# LIST OF FIGURES


| FIGURE | | PAGE |
|---|---|---|
| 5-1 | Flowchart Of A Program Loop | 5-2 |
| 5-2 | A Program Loop Which Allows Zero Iterations | 5-3 |
| 7-1 | 7-Segment Arrangement | 7-4 |
| 11-1 | An Output Demultiplexer Controlled By A Counter | 11-3 |
| 11-2 | An Output Demultiplexer Controlled By A Port | 11-3 |
| 11-3 | An Input Multiplexer Controlled By A Counter | 11-4 |
| 11-4 | An Input Multiplexer Controlled By A Port | 11-4 |
| 11-5 | An Input Handshake | 11-6 |
| 11-6 | An Output Handshake | 11-7 |
| 11-7 | A Pushbutton Circuit | 11-11 |
| 11-8 | A Toggle Switch Circuit | 11-17 |
| 11-9 | A Debouncing Circuit Based On Cross-Coupled NAND Gates | 11-17 |
| 11-10 | A Multiple-Position Switch | 11-21 |
| 11-11 | A Multiple-Position Switch With An Encoder | 11-22 |
| 11-12 | Interfacing an LED | 11-27 |
| 11-13 | Interfacing a 7-Segment Display | 11-30 |
| 11-14 | Display Organization | 11-31 |
| 11-15 | 7-Segment Representations of Decimal Digits | 11-32 |
| 11-16 | A Small Keyboard | 11-41 |
| 11-17 | A Keyboard Matrix | 11-41 |
| 11-18 | I/O Arrangement For A Keyboard Scan | 11-43 |
| 11-19 | I/O Interface For An Encoded Keyboard | 11-48 |
| 11-20 | The 8212 I/O Port | 11-49 |
| 11-21 | The 8212 I/O Port as an Interrupting Input Port | 11-51 |
| 11-22a | Signetics NE5018 D/A Converter | 11-53 |
| 11-22b | NE5018 Connection to 8080 System | 11-54 |
| 11-23 | Interface for an 8-Bit Digital-to-Analog Converter | 11-54 |
| 11-24 | The 8212 I/O Port As A Latched Output Port | 11-54 |
| 11-25 | Teledyne 8703 A/D Converter | 11-58 |
| 11-26 | Interface for an 8-Bit Analog-to-Digital Converter | 11-59 |
| 11-27 | Teletypewriter Data Format | 11-61 |
| 11-28 | Flowchart for Receive Procedure | 11-62 |
| 11-29 | Flowchart for Transmit Procedure | 11-64 |
| 12-1 | Using Pullup Resistors to Form the RST 7 Instruction | 12-4 |
| 12-2 | Using the 8228 System Controller to Form the RST 7 Instruction | 12-5 |
| 12-3 | Using an 8212 I/O Port to Form the RST 7 Instruction | 12-5 |
| 12-4 | Forming Eight RST Instructions with a Priority Encoder | 12-5 |
| 12-5 | The 8214 Priority Interrupt Control Unit | 12-8 |
| 12-6 | The 8214 Priority Interrupt Control Unit Used as an 8-Level Controller | 12-10 |
| 12-7 | The 8259 Programmable Interrupt Controller | 12-11 |
| 12-8 | A Single Input Interrupt | 12-12 |

# LIST OF FIGURES (Continued)


| FIGURE | | PAGE |
|---|---|---|
| 13-1 | Flowchart of Software Development | 13-2 |
| 13-2 | The Switch and Light System | 13-6 |
| 13-3 | The Switch-Based Memory Loader | 13-8 |
| 13-4 | Block Diagram of a Verification Terminal | 13-10 |
| 13-5 | Verification Terminal Keyboard | 13-11 |
| 13-6 | Verification Terminal Display | 13-11 |
| 13-7 | Standard Flow Diagram Symbols | 13-15 |
| 13-8 | Flowchart of a One-Second Response to a Switch | 13-17 |
| 13-9 | Flowchart of Switch-Based Memory Loader | 13-18 |
| 13-10 | Flowchart of Keyboard Entry Process | 13-19 |
| 13-11 | Flowchart of Keyboard Entry Process with SEND Key | 13-20 |
| 13-12 | Flowchart of Keyboard Entry Process with Function Keys | 13-21 |
| 13-13 | Flowchart of Receive Routine | 13-22 |
| 13-14 | Flowchart of an Unstructured Program | 13-26 |
| 13-15 | Flowchart of the If-Then-Else Structure | 13-27 |
| 13-16 | Flowchart of the Do-While Structure | 13-27 |
| 13-17 | Initial Flowchart for Transaction Terminal | 13-39 |
| 13-18 | Flowchart for Expanded KEYBOARD Routine | 13-40 |
| 14-1 | A Simple Breakpoint Routine | 14-2 |
| 14-2 | Flowchart of Register Dump Program | 14-4 |
| 14-3 | Results of a Typical Register Dump | 14-5 |
| 14-4 | Results of a Typical Memory Dump | 14-5 |
| 14-5 | Flowchart of Decimal to 7-Segment Conversion | 14-13 |
| 14-6 | Flowchart of Sort Program | 14-16 |
| 16-1 | I/O Configuration | 16-2 |
| 16-2 | I/O Configuration | 16-16 |
| 16-3 | Analog Hardware | 16-17 |
| 16-4 | Thermistor Characteristics (Fenwal GA51J1 Bead) | 16-18 |
| 16-5 | Typical E-I Curve for Thermistor (25°C) | 16-18 |
| 16-6 | Generating an Internal Clock Frequency | 16-19 |

# LIST OF TABLES


| TABLE | | PAGE |
|---|---|---|
| 1-1 | Hexadecimal Conversion Table | 1-4 |
| 2-1 | The Fields of an Assembly Language Instruction | 2-1 |
| 2-2 | Standard 8080 Assembler Delimiters | 2-2 |
| 2-3 | Assigning and Using a Label | 2-3 |
| 3-1 | Frequently Used Instructions of the 8080A and 8085 | 3-5 |
| 3-2 | Occasionally Used Instructions of the 8080A and 8085 | 3-6 |
| 3-3 | Seldom Used Instructions of the 8080A and 8085 | 3-7 |
| 3-4 | A Summary of 8080A/8085 Microcomputer Instruction Set | 3-10 |
| 3-5 | A Summary of Instruction Object Codes and Execution Cycles | 3-18 |
| 6-1 | 7-Bit ASCII Character Code Chart | 6-2 |
| 11-1 | Data Input vs. Switch Position | 11-22 |
| 11-2 | 7-Segment Representations of Decimal Numbers | 11-33 |
| 11-3 | 7-Segment Representation of Letters and Symbols | 11-34 |
| 11-4 | Comparison Between Independent Connections and Matrix Connections for Keyboards | 11-42 |
| 12-1 | The Restart Instruction | 12-3 |
| 12-2 | RST Instructions Produced by the Various Request Inputs | 12-7 |
| 12-3 | Interrupts Allowed for Various Status Register Values | 12-7 |
| 14-1 | Intel 8080 Restart Addresses | 14-3 |
| 16-1 | Input Connections for Timer Keyboard | 16-2 |
| 16-2 | Output Connections for Timer Keyboard | 16-2 |

# QUICK INDEX


| INDEX | | PAGE |
|---|---|---|
| A | ADDRESS FIELD, NUMBERS AND CHARACTERS IN | 3-101 |
| | ALGEBRAIC NOTATION | 1-9 |
| | ALLOCATING RAM | 2-7 |
| | ARITHMETIC AND LOGICAL EXPRESSIONS | 2-10 |
| | ASCII, HANDLING DATA IN | 6-1 |
| | ASCII CHARACTERS | 2-10 |
| | ASSEMBLER | 1-6 |
| | ASSEMBLER, CHOOSING AN | 1-6 |
| | ASSEMBLER ARITHMETIC AND LOGICAL OPERATIONS | 3-101 |
| | ASSEMBLER DIRECTIVE | 2-4 |
| | ASSEMBLER OPERATIONS, ORDER OF | 3-102 |
| | ASSEMBLY LANGUAGE, APPLICATIONS FOR | 1-11 |
| | ASSEMBLY LANGUAGE FIELDS | 2-1 |
| | ASSEMBLY LANGUAGE PROGRAM | 1-5 |
| B | BASIC SOFTWARE DELAY | 11-8 |
| | BETTER ALGORITHMS | 15-14 |
| | BINARY, DECIMAL ACCURACY IN | 8-3 |
| | BINARY INSTRUCTIONS | 1-1 |
| | BLANKING A LEADING ZERO | 16-22 |
| | BOOTSTRAP LOADER | 2-15 |
| | BOTTOM-UP DESIGN | 13-35 |
| | BREAKPOINT | 14-2 |
| C | CHANGING THE RETURN ADDRESS | 12-15 |
| | CHECKLIST, WHAT TO INCLUDE IN | 14-8 |
| | CHOOSING A TIMING METHOD | 11-8 |
| | CHOOSING LABELS | 2-3 |
| | CHOOSING USEFUL NAMES | 15-2 |
| | CODING | 13-3 |
| | CODING, RELATIVE IMPORTANCE OF | 13-1 |
| | COMBINING CONTROL INFORMATION | 11-39 |
| | COMMENTING, QUESTIONS FOR | 15-4 |
| | COMMENTING EXAMPLES | 15-4 |
| | COMMENTING GUIDELINES | 15-2 |
| | COMMENTING TECHNIQUES | 2-13 |
| | COMMON-ANODE AND COMMON-CATHODE DISPLAYS | 11-30 |
| | COMMON ERRORS | 14-9 |
| | COMPILER | 1-7 |
| | COMPILERS, COST OF | 1-9 |
| | COMPUTER PROGRAM | 1-2 |
| | CONTROL AND STATUS INFORMATION | 11-38 |
| | CORRECTING KEYBOARD ERRORS | 13-13 |
| | CORRECTING TRANSMISSION ERRORS | 13-13 |
| | CROSS-ASSEMBLER | 2-14 |

# QUICK INDEX (Continued)


| INDEX | | PAGE |
|---|---|---|
| D | DATA, FORMING CLASSES OF | 14-23 |
| | DEBOUNCING IN SOFTWARE | 11-14 |
| | DEBOUNCING WITH CROSS-COUPLED NAND GATES | 11-17 |
| | DEBUGGING | 13-3 |
| | DEBUGGING, USING TEST CASES FROM | 14-21 |
| | DEBUGGING A CODE CONVERSION PROGRAM | 14-12 |
| | DEBUGGING A SORT PROGRAM | 14-15 |
| | DEBUGGING INTERRUPT-DRIVEN PROGRAMS | 14-11 |
| | DECIDING ON A MAJOR CHANGE | 15-15 |
| | DECIMAL ADJUST | 8-6 |
| | DECIMAL DATA OR ADDRESSES | 2-9 |
| | DEFINING A SWITCH-BASED MEMORY LOADER | 13-7 |
| | DEFINING A VERIFICATION TERMINAL | 13-10 |
| | DEFINING NAMES | 2-6 |
| | DEFINING SWITCH AND LIGHT SYSTEM | 13-6 |
| | DEFINITION LIST, RULES FOR | 15-7 |
| | DEFINITIONS, PLACEMENT OF | 2-7 |
| | DELIMITERS | 2-2 |
| | DIRECT MEMORY ACCESS | 11-5 |
| | DIVISION ALGORITHM | 8-11 |
| | DOCUMENTATION | 13-3 |
| | DOCUMENTATION PACKAGE | 15-12 |
| | DOCUMENTING STATUS AND CONTROL TRANSFERS | 11-40 |
| | DOCUMENTING SUBROUTINES | 10-2 |
| | DOUBLE BUFFERING | 12-16 |
| E | 8-BIT SUMMATION | 5-4 |
| | 8080 INTERRUPT RESPONSE | 12-3 |
| | 8080A DELAY LOOP CONSTANT | 11-10 |
| | 8085 DELAY LOOP CONSTANT | 11-11 |
| | 8085 INTERRUPT SYSTEM | 12-6 |
| | 8214 INTERRUPT CONTROL UNIT | 12-6 |
| | 8259 INTERRUPT CONTROLLER | 12-9 |
| | EMPTYING A BUFFER WITH INTERRUPTS | 12-18 |
| | ENABLING AND DISABLING INTERRUPTS | 12-2 |
| | ENCODED KEYBOARDS | 11-48 |
| | EQU PSEUDO-OPERATION | 3-100 |
| | ERROR CONSIDERATIONS | 13-5 |
| | EXAMPLE FORMAT | 4-1 |
| | EXAMPLES, GUIDELINES FOR | 4-1 |
| | EXPANDING PROGRAM STUBS | 13-36 |
| | EXPANDING THE KEYBOARD ROUTINE | 13-38 |
| F | FILLING A BUFFER VIA INTERRUPTS | 12-16 |
| | FLOWCHARTING, ADVANTAGES OF | 13-15 |
| | FLOWCHARTING, DISADVANTAGES OF | 13-16 |
| | FLOWCHARTING SECTIONS | 13-19 |
| | FLOWCHARTING SWITCH AND LIGHT SYSTEM | 13-16 |
| | FLOWCHARTING THE CREDIT VERIFICATION | 13-19 |
| | FLOWCHARTING THE SWITCH-BASED MEMORY LOADER | 13-17 |
| | FLOWCHARTS, HINTS FOR USING | 15-6 |
| | FORMAT | 2-2 |
| | FORTRAN | 1-7 |

# QUICK INDEX (Continued)


| INDEX | | PAGE |
|---|---|---|
| G | GENERAL SERVICE ROUTINES, TASKS FOR | 12-24 |
| H | HAND ASSEMBLY | 1-5 |
| | HAND-CHECKING QUESTIONS | 14-9 |
| | HANDSHAKE | 11-2 |
| | HASHING | 9-4 |
| | HEXADECIMAL LOADER | 1-4 |
| | HIGH-LEVEL LANGUAGES, ADVANTAGES OF | 1-9 |
| | HIGH-LEVEL LANGUAGE, APPLICATIONS FOR | 1-11 |
| | HIGH-LEVEL LANGUAGES, DISADVANTAGES OF | 1-9 |
| | HIGH-LEVEL LANGUAGES, INEFFICIENCY OF | 1-9 |
| | HIGH-LEVEL LANGUAGES, MACHINE INDEPENDENCE OF | 1-8 |
| | HIGH-LEVEL LANGUAGES, OVERHEAD FOR | 1-10 |
| | HIGH-LEVEL LANGUAGES, PORTABILITY OF | 1-8 |
| | HIGH-LEVEL LANGUAGES, SYNTAX OF | 1-8 |
| | HIGH-LEVEL LANGUAGES, UNSUITABILITY OF | 1-10 |
| I | IDENTIFYING KEY CLOSURES | 11-44 |
| | INITIALIZING RAM | 2-8 |
| | INPUT, FACTORS IN | 13-4 |
| | INSTRUCTION EXECUTION, STATUS CHANGES WITH | 3-9 |
| | INSTRUCTIONS, DEFINING A SEQUENCE OF | 2-11 |
| | INTERFACING SLOW DEVICES | 11-2 |
| | INTERRUPT HANDLING BY MONITORS | 12-13 |
| | INTERRUPT SERVICE ROUTINE, LOCATION OF | 12-13 |
| | INTERRUPT SYSTEMS, CHARACTERISTICS OF | 12-1 |
| | INTERRUPTS, DISADVANTAGES OF | 12-2 |
| | INTERRUPTS, REASONING BEHIND | 12-1 |
| | I/O AND MEMORY | 11-1 |
| | I/O CATEGORIES | 11-1 |
| | I/O DEVICES, SYNCHRONIZING WITH | 11-38 |
| J | JUMP INSTRUCTIONS, LABELS IN | 2-2 |
| K | KEY TABLE | 16-6 |
| | KEYBOARD INTERRUPT | 12-14 |
| | KEYBOARD SCAN | 11-42 |
| L | LABEL FIELD | 2-2 |
| | LABELING, RULES OF | 2-4 |
| | LANGUAGE LEVELS, APPLICATION AREAS FOR | 1-10 |
| | LANGUAGE LEVELS, FUTURE TRENDS IN | 1-12 |
| | LINKING LOADERS | 2-15 |
| | LOCAL OR GLOBAL VARIABLES | 2-13 |
| | LOCATION COUNTER | 2-7 |
| | LOGIC ANALYZER | 14-7 |
| | LOGIC ANALYZERS, IMPORTANT FEATURES OF | 14-8 |

# QUICK INDEX (Continued)


| INDEX | | PAGE |
|---|---|---|
| **M** | MACHINE LANGUAGE, APPLICATIONS FOR | 1-11 |
| | MACHINE LANGUAGE PROGRAM | 1-2 |
| | MACROASSEMBLER | 2-14 |
| | MACROS, ADVANTAGES OF | 2-13 |
| | MACROS, DISADVANTAGES OF | 2-13 |
| | MAINTAINING REAL TIME | 12-21 |
| | MAINTENANCE AND RE-DESIGN | 13-3 |
| | MAJOR OR MINOR REORGANIZATION | 15-13 |
| | MATRIX KEYBOARD | 11-40 |
| | MEASURING PROGRESS IN STAGES | 13-1 |
| | MEMORY DUMP | 14-5 |
| | MEMORY LOADER ERROR HANDLING | 13-9 |
| | META-ASSEMBLER | 2-14 |
| | MICROASSEMBLER | 2-14 |
| | MNEMONICS, PROBLEM WITH | 1-5 |
| | MODULAR PROGRAMMING, ADVANTAGES OF | 13-23 |
| | MODULAR PROGRAMMING, DISADVANTAGES OF | 13-23 |
| | MODULAR PROGRAMMING, RULES FOR | 13-25 |
| | MODULARIZING THE SWITCH AND LIGHT SYSTEM | 13-24 |
| | MODULARIZING THE SWITCH-BASED MEMORY LOADER | 13-24 |
| | MODULARIZING THE VERIFICATION TERMINAL | 13-24 |
| | MULTIPLICATION ALGORITHM | 8-7 |
| **N** | NAMES, CHOICE OF | 2-6 |
| | NAMES, USE OF | 2-6 |
| | NON-MASKABLE INTERRUPT | 12-2 |

# QUICK INDEX (Continued)


| INDEX | | PAGE |
|---|---|---|
| O | OBJECT PROGRAM | 1-2,1-6 |
| | OCTAL OR HEXADECIMAL | 1-3 |
| | ONE-PASS ASSEMBLER | 2-14 |
| | OPERATOR ERROR CORRECTION IN MEMORY LOADER | 13-9 |
| | OPERATOR INTERACTION | 13-5 |
| | ORG PSEUDO-OPERATION | 3-100 |
| | OTHER MAJOR CHANGES | 15-14 |
| | OTHER NUMBER SYSTEMS | 2-9 |
| | OTHER SORTING METHODS | 9-14 |
| P | PASSING PARAMETERS | 10-1 |
| | POLLING | 12-2 |
| | PORTABILITY | 1-7 |
| | PRINTER INTERRUPT | 12-17 |
| | PRIORITY | 12-2 |
| | PROBLEM DEFINITION | 13-3 |
| | PROCESSING, FACTORS IN | 13-4 |
| | PRODUCING A SINGLE RESTART INSTRUCTION | 12-4 |
| | PROGRAM DESIGN | 13-3 |
| | PROGRAM DESIGN, BASIC PRINCIPLES OF | 13-14 |
| | PROGRAM STUBS | 13-35 |
| | PROGRAMMING GUIDELINES | 4-2 |
| R | REAL-TIME CLOCK | 12-19 |
| | REAL-TIME CLOCK, FREQUENCY OF | 12-19 |
| | REAL-TIME CLOCK, PRIORITY OF | 12-19 |
| | REAL-TIME CLOCK, SYNCHRONIZATION WITH | 12-19 |
| | RE-DESIGN, COST OF | 15-12 |
| | REDUCING TRANSMISSION ERRORS | 11-5 |
| | RE-ENTRANT SUBROUTINE | 10-2 |
| | REGISTER DUMP | 14-3 |
| | RELOCATING LOADER | 2-15 |
| | RELOCATION | 10-2 |
| | RESIDENT ASSEMBLER | 2-14 |
| | RESTART INSTRUCTION | 12-3 |
| | RIM INSTRUCTION | 12-6 |
| | ROLLOVER | 11-48 |
| | RST AS A BREAKPOINT | 14-2 |

# QUICK INDEX (Continued)


| INDEX | | PAGE |
|---|---|---|
| S | SAVING AND RESTORING REGISTERS AND PRIORITY | 12-26 |
| | SAVING EXECUTION TIME | 15-13 |
| | SAVING MEMORY | 15-13 |
| | SEARCHING METHODS | 9-7 |
| | SELECTING TEST DATA FROM CLASSES | 14-23 |
| | SELF-DOCUMENTING PROGRAMS, RULES FOR | 15-1 |
| | SEPARATING STATUS INFORMATION | 11-39 |
| | 7-SEGMENT REPRESENTATIONS | 11-32 |
| | SIM INSTRUCTION | 12-6 |
| | SIMPLE SORTING ALGORITHM | 9-10 |
| | SIMULATOR | 14-6 |
| | SINGLE-STEP | 14-1 |
| | SINGLE-STEP MODE, LIMITATIONS OF | 14-2 |
| | SOFTWARE DEVELOPMENT, STAGES OF | 13-1 |
| | SOURCE PROGRAM | 1-6 |
| | SPECIAL INSTRUCTIONS | 4-3 |
| | STANDARD PROGRAM LIBRARY FORMS | 15-9 |
| | STANDARD TTY CHARACTER FORMAT | 11-60 |
| | START BIT INTERRUPT | 12-24 |
| | STARTUP INTERRUPT | 12-11 |
| | STATUS CONDITIONS | 3-71 |
| | STOPWATCH INPUT PROCEDURE | 16-1 |
| | STROBE | 11-5 |
| | STRUCTURED KEYBOARD ROUTINE | 13-31 |
| | STRUCTURED PROGRAM FOR THE CREDIT-VERIFICATION TERMINAL | 13-31 |
| | STRUCTURED PROGRAMMING, ADVANTAGES OF | 13-28 |
| | STRUCTURED PROGRAMMING, BASIC STRUCTURES OF | 13-26 |
| | STRUCTURED PROGRAMMING, DISADVANTAGES OF | 13-29 |
| | STRUCTURED PROGRAMMING, RULES FOR | 13-35 |
| | STRUCTURED PROGRAMMING, WHEN TO USE | 13-29 |
| | STRUCTURED PROGRAMMING FOR THE SWITCH-BASED MEMORY LOADER | 13-30 |
| | STRUCTURED PROGRAMMING IN THE SWITCH AND LIGHT SYSTEM | 13-29 |
| | STRUCTURED RECEIVE ROUTINE | 13-33 |
| | STRUCTURED TESTING | 14-22 |
| | STRUCTURES, EXAMPLES OF | 13-28 |
| | STRUCTURES, TERMINATORS FOR | 13-35 |
| | SUBROUTINE CALL USING RST | 3-80 |
| | SUBROUTINE INSTRUCTIONS | 10-1 |
| | SUBROUTINE LIBRARY | 10-1 |
| | SWITCH AND LIGHT ERROR HANDLING | 13-7 |
| | SWITCH AND LIGHT INPUT | 13-6 |
| | SWITCH AND LIGHT OUTPUTS | 13-6 |
| | SWITCH BOUNCE | 11-14 |
| | SYMBOL TABLE | 2-6 |

# QUICK INDEX (Continued)


| INDEX | | PAGE |
|---|---|---|
| T | TELETYPEWRITER INTERRUPT | 12-23 |
| | TESTING | 13-3 |
| | TESTING, RULES FOR | 14-24 |
| | TESTING A SORT PROGRAM | 14-23 |
| | TESTING AN ARITHMETIC PROGRAM | 14-24 |
| | TESTING AIDS | 14-22 |
| | TESTING SPECIAL CASES | 14-23 |
| | THERMOMETER ANALOG HARDWARE | 16-15 |
| | TIMING INTERVALS, METHODS FOR PRODUCING | 11-8 |
| | TIMING INTERVALS, USES OF | 11-8 |
| | TOP-DOWN DESIGN, ADVANTAGES OF | 13-36 |
| | TOP-DOWN DESIGN, DISADVANTAGES OF | 13-36 |
| | TOP-DOWN DESIGN, FORMAT FOR | 13-39 |
| | TOP-DOWN DESIGN METHODS | 13-35 |
| | TOP-DOWN DESIGN OF SWITCH AND LIGHT SYSTEM | 13-36 |
| | TOP-DOWN DESIGN OF SWITCH-BASED MEMORY LOADER | 13-37 |
| | TOP-DOWN DESIGN OF VERIFICATION TERMINAL | 13-38 |
| | TRANSPARENT DELAY ROUTINE | 11-8 |
| | TTY INTERFACE | 11-60 |
| | TTY RECEIVE MODE | 11-61 |
| | TTY RECEIVE PROGRAM | 11-63 |
| | TTY TRANSMIT MODE | 11-65 |
| | TTY TRANSMIT PROGRAM | 11-65 |
| | TWO-PASS ASSEMBLER | 2-14 |
| | TYPICAL DEFINITION LIST | 15-8 |
| | TYPICAL MEMORY MAP | 15-7 |
| U | UART | 11-65 |
| | UART INTERRUPTS | 12-23 |
| | USING A CALIBRATION TABLE | 16-20 |
| | USING A DIGITAL ENCODER | 11-22 |
| | USING REGISTER M | 4-2 |
| | USING RST 0 | 12-25 |
| | USING THE ACCUMULATOR | 4-2 |
| V | VECTORING | 12-2 |
| | VERIFICATION TERMINAL ERROR HANDLING | 13-12 |
| | VERIFICATION TERMINAL INPUTS | 13-11 |
| | VERIFICATION TERMINAL OUTPUTS | 13-12 |
| W | WAITING FOR A KEY CLOSURE | 11-42 |

# Chapter 1
# INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

**This book describes assembly language programming. It assumes that you are familiar with An Introduction To Microcomputers: Volume I — Basic Concepts (particularly Chapters 6 and 7). This book does not discuss the general features of computers, microcomputers, addressing methods, or instruction sets; you should refer to An Introduction To Microcomputers: Volume I for that information.**

## HOW THIS BOOK HAS BEEN PRINTED

**Notice that text in this book has been printed in boldface type and lightface type. This has been done to help you skip those parts of the book that cover subject matter with which you are familiar. You can be sure that lightface type only expands on information presented in the previous boldface type.** Therefore, only read boldface type until you reach a subject about which you want to know more, at which point start reading the lightface type.

## THE MEANING OF INSTRUCTIONS

**The instruction set of a microprocessor is simply the set of binary inputs which produce defined actions during an instruction cycle.** An instruction set is to a microprocessor what a function table is to a logic device such as a gate, adder, or Shift register. Of course, the actions which the microprocessor performs in response to the instruction inputs are far more complex than the actions which combinatorial logic devices perform in response to their inputs.

**BINARY INSTRUCTIONS**

**An instruction is simply a binary bit pattern — it must be present at the data inputs to the microprocessor at the proper time in order to be interpreted as an instruction.** For example, in the case of the 8080 microprocessor, when the 8-bit binary pattern 10000000 is the input during an instruction fetch operation, it means:

"Add the contents of Register B to the contents of the Accumulator".

Similarly, the pattern 00111110 means:

"Place the contents of the next word of program memory in the Accumulator".

The microprocessor (like any other computer) only recognizes binary patterns as instructions or data; it does not recognize words or octal, decimal, or hexadecimal numbers.

## A COMPUTER PROGRAM

**A program is a sequence of instructions which cause a computer to perform a particular task.**

**COMPUTER PROGRAM**

Actually, a computer program includes more than instructions; it also contains the data and memory addresses which the microprocessor needs to accomplish the task defined by the instructions. Clearly, if the microprocessor is to perform an addition, it must have two numbers to add and a destination for the result. The computer program must somehow determine the sources of the data and the destination of the result as well as the operation.

All microprocessors execute instructions sequentially unless one of the instructions changes the execution sequence or halts the computer. i.e., the processor gets the next instruction from the next consecutive memory address unless the present instruction specifically directs it to do otherwise.

**Ultimately every program becomes a set of binary numbers. For example, this is the 8080 program which adds the contents of memory locations $60_{16}$ and $61_{16}$ and places the result in memory location $62_{16}$:**

```
00111010
01100000
00000000
01000111
00111010
01100001
00000000
10000000
00110010
01100010
00000000
```

**OBJECT PROGRAM**

**MACHINE LANGUAGE PROGRAM**

**This is a machine language, or object program.** If this program were entered into the memory of an 8080-based microcomputer, the microcomputer would be able to execute it directly.

## THE PROGRAMMING PROBLEM

**There are many difficulties associated with creating programs as object, or binary machine language, programs. These are some of the problems:**

1) The programs are difficult to understand or debug (binary patterns all look the same, particularly after you have looked at them for a few hours).
2) The programs are slow to enter since you must enter each bit individually.
3) The programs do not describe the task which you want the computer to perform in anything resembling a human readable format.
4) The programs are long and tiresome to write.
5) The programmer often makes careless errors which are subsequently very difficult to find.

For example, the **following version of the addition object program contains a single bit error. Try to find it:**

```
00111010
01100000
00000000
01000111
01110010
01100001
00000000
10000000
00110010
01100010
00000000
```

Although the computer handles binary numbers with ease, people do not. People find binary programs long, tiresome, confusing, and meaningless. Eventually, a programmer may start remembering some of the binary codes, but such effort should be spent more productively.

## USING OCTAL OR HEXADECIMAL

OCTAL OR HEXADECIMAL

**We can improve the situation somewhat by writing instructions using octal or hexadecimal, rather than binary numbers.** We will use hexadecimal numbers in this book because they are shorter, and because they are the standard for the microprocessor industry. Table 1-1 defines the hexadecimal digits and their binary equivalents. **The 8080 program to add two numbers now becomes:**

```
3A
60
00
47
3A
61
00
80
32
62
00
```

At the very least, the hexadecimal version is shorter to write and not quite so tiring to examine.

**Errors are somewhat easier to find in a sequence of hexadecimal digits. The erroneous version of the addition program, in hexadecimal form becomes:**

```
3A
60
00
47
72
61
00
80
32
62
00
```

**The mistake is easier to spot.**

**What do we do with this hexadecimal program? The microprocessor only understands binary instruction codes.** The answer is that we must convert the hexadecimal numbers to binary numbers. This conversion is a repetitive, tiresome task. People who attempt it make all sorts of petty mistakes, such as looking at the wrong line, dropping a bit, or transposing a bit or a digit.

**HEXADECIMAL LOADER**

This repetitive, grueling task is, however, a perfect job for a computer. The computer never gets tired or bored and never makes silly mistakes. **The idea then is to write a program which takes hexadecimal numbers and converts them into binary numbers. This is a standard program provided with many microprocessors; it is called a "hexadecimal loader."**

Is a hexadecimal loader worth having? If you are willing to write a program using binary numbers, and you are prepared to enter the program in its binary form into the computer, then you will not need the hexadecimal loader.

If you choose the hexadecimal loader, you will have to pay a price for it. The hexadecimal loader is itself a program which you must load into memory. Furthermore, the hexadecimal loader will occupy y memory — memory which you may want to use in some other way.

The basic tradeoff, therefore, is the cost and memory requirements of the hexadecimal loader versus the savings in programmer time.

A hexadecimal loader is well worth its small cost.

Table 1-1. Hexadecimal Conversion Table

| Hexadecimal Digit | Binary Equivalent | Decimal Equivalent |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| A | 1010 | 10 |
| B | 1011 | 11 |
| C | 1100 | 12 |
| D | 1101 | 13 |
| E | 1110 | 14 |
| F | 1111 | 15 |

A hexadecimal loader certainly does not solve every programming problem. The hexadecimal version of the program is still difficult to read or understand; for example, it does not distinguish instructions from data or addresses, nor does the program listing provide any suggestion as to what the program does. What does 32 or 47 or 3A mean? Memorizing a card full of codes hardly is an appetizing proposition. Furthermore, the codes will be entirely different for a different microprocessor and the program will require a large amount of documentation.

## INSTRUCTION CODE MNEMONICS

**An obvious programming improvement is to assign a name to each instruction code. The instruction code name is called a "mnemonic", or memory jogger. The instruction mnemonic should describe in some way what the instruction does.**

**PROBLEM WITH MNEMONICS**

In fact, every microprocessor manufacturer (they can't remember hexadecimal codes either) provides a set of mnemonics for the microprocessor instruction set. But **you do not have to abide by the manufacturer's mnemonics;** there is nothing sacred about them. However, they are standard for a given microprocessor, and therefore understood by all users. These are the instruction names you will find in manuals, cards, books, articles, and programs. The problem with selecting instruction mnemonics is that not all instructions have "obvious" names. Some instructions do have obvious names (e.g., ADD, AND, OR), others have obvious contractions (e.g., SUB for subtraction, XOR for exclusive OR), while still others have neither. The result is such mnemonics as WMP, PCHL, and even SOB (try and guess what that means!) Most manufacturers come up with some reasonable names and some hopeless ones. However, users who devise their own mnemonics rarely seem to do much better than the manufacturer.

Along with the instruction mnemonics, the manufacturer will usually assign names to the CPU's registers. As with the instruction names, some register names are obvious (e.g., A for Accumulator) while others may have only historical significance. Again, we will use the manufacturer's suggestions simply to promote standardization.

**ASSEMBLY LANGUAGE PROGRAM**

**If we use standard 8080 instruction and register mnemonics, as defined by Intel, our 8080 addition program becomes:**

```
LDA
60
00
MOV B,A
LDA
61
00
ADD B
STA
62
00
```

The program is still far from obvious but at least some parts are comprehensible. ADD B is a considerable improvement over 80; LDA, MOV, and STA do suggest loading, moving, and storing respectively. We now know which lines are instructions and which are data or addresses. **Such a program is an assembly language program.**

## THE ASSEMBLER PROGRAM

**HAND ASSEMBLY**

How do we get the assembly language program into the computer? We have to translate it, either into hexadecimal or into binary numbers. **You can translate an assembly language program by hand,** instruction by instruction. This is called hand assembly.

Hand assembly of a three-instruction sequence may be illustrated as follows:

| Instruction Name | Hexadecimal Equivalent |
|---|---|
| ADD B | 80 |
| LDA | 3A |
| STA | 32 |

As in the case of hexadecimal to binary conversion, hand assembly is a rote task which is uninteresting, repetitive, and subject to numerous minor errors. Picking the wrong line, transposing digits, omitting instructions, and misreading the codes are only a few of the mistakes you may make. Most microprocessors complicate the task even further by having instructions with different word lengths. Some instructions are one word long while other instructions are two or three words long. Some instructions require data in the second and third words, others require memory addresses, register numbers, or who knows what?

**ASSEMBLER**
**SOURCE PROGRAM**
**OBJECT PROGRAM**

**Assembly is another rote task which we can assign to the microcomputer. The microcomputer never makes any mistakes when translating codes; it always knows how many words and what format each instruction requires. The program which does this job is called an "assembler". The assembler program translates a user program, or "source" program written with mnemonics, into a machine language program, or "object" program, which the microcomputer can understand. The assembler's input is a source program and its output is an object program.**

**The tradeoffs we discussed in connection with the hexadecimal loader are magnified in the case of the assembler.** Assemblers are more expensive, occupy more memory, and require more peripherals and execution time than do hexadecimal loaders. While users may (and often do) write their own loaders, few care to write their own assemblers.

**Assemblers have their own rules which you must learn to abide by.** These include the use of certain markers (such as spaces, commas, semi-colons, or colons) in appropriate places, correct spelling, the proper control information and perhaps even the correct placement of names and numbers. These rules typically are a minor hindrance which can be quickly overcome.

## ADDITIONAL FEATURES OF ASSEMBLERS

Early assembler programs did little more than translate the mnemonic names of instructions and registers into their binary equivalents. However, most assemblers now provide such additional features as:

1) Allowing the user to assign names to memory locations, input and output devices, and even sequences of instructions.
2) Converting data or addresses from various number systems (e.g., decimal or hexadecimal) to binary and converting characters into their ASCII or EBCDIC binary codes.
3) Performing some arithmetic as part of the assembly process.
4) Telling the loader program where in memory parts of the program or data should be placed.
5) Allowing the user to assign areas of memory as temporary data storage, and to place fixed data in areas of program memory.
6) Providing the information required to include standard programs from program libraries, or programs written at some other time, in the current program.
7) Allowing the user to control the format of the program listing and the input and output devices employed.

**CHOOSING AN ASSEMBLER**

All of these features, of course, involve additional cost and memory. Microcomputers generally have much simpler assemblers than do larger computers, but the tendency always is for the size of assemblers to increase. You will often have a choice of assemblers. The important criterion is not how many off-beat features the assembler has, but rather how convenient it is to work with in normal practice.

## DISADVANTAGES OF ASSEMBLY LANGUAGE

**The assembler, like the hexadecimal loader, does not solve all the problems of programming. One problem is the tremendous gap between the microcomputer instruction set and the tasks which the microcomputer is to perform.** Computer instructions tend to do things like add the contents of two registers, shift the contents of the Accumulator one bit, or place a new value in the Program Counter. On the other hand, a user generally wants a microcomputer to do something like check if an analog reading has exceeded a threshold, look for and react to a particular command from a teletypewriter, or activate a relay at the proper time. An assembly language programmer must translate such tasks into a sequence of simple computer instructions. The translation can be a difficult, time-consuming job.

Furthermore, **if you are programming in assembly language, you must have detailed knowledge of the particular microcomputer you are using.** You must know what registers and instructions the microcomputer has, precisely how the instructions affect the various registers, what addressing methods the computer uses, and a myriad of other information. None of this information is relevant to the task which the microcomputer must ultimately perform.

PORTABILITY

**In addition, assembly language programs are not portable.** Each microcomputer has its own assembly language, which reflects its own architecture. An assembly language program written for the 8080 will not run on the Motorola 6800, the Fairchild F8, or the National Semiconductor PACE. For example, the addition program written for the Motorola 6800 would be:

```
LDAA    $60
ADDA    $61
STAA    $62
```

The lack of portability not only means that you won't be able to use your assembly language program on another microcomputer, but it also means that you won't be able to use any programs that weren't specifically written for the microcomputer you are using. This is a particular drawback for microcomputers, since these devices are new and few assembly language programs exist for them. The result, too frequently, is that you are on your own. If you need a program to perform a particular task, you are not likely to find it in the small program libraries that most manufacturers provide. Nor are you likely to find it in an archive, journal article, or someone's old program file. You will probably have to write it yourself.

## HIGH-LEVEL LANGUAGES

COMPILER

**The solution to many of the difficulties associated with assembly language programs is to use, instead, "high-level" or "procedure-oriented" languages. Such languages allow you to describe tasks in forms that are problem oriented rather than computer oriented. Each statement in a high-level language performs a recognizable function; it will generally correspond to many assembly language instructions. A program called a compiler translates the high-level language source program into object or machine language instructions.**

FORTRAN

Many different high-level languages exist for different types of tasks. If, for example, you can express what you want the computer to do in algebraic notation, you can write your program in FORTRAN (Formula Translation Language), the oldest and most widely used of the high-level languages. Now, if you want to add two numbers, you just tell the computer:

```
SUM = NUMB1+NUMB2
```

That is a lot simpler (and a lot shorter) than either the equivalent machine language program or the equivalent assembly language program. Other high-level languages include COBOL (for business applications), ALGOL (another algebraic language), PL/I (a combination of FORTRAN, ALGOL, and COBOL), and APL and BASIC (languages that are popular for time-sharing systems).

## ADVANTAGES OF HIGH-LEVEL LANGUAGES

**Clearly high-level languages make programs easier and faster to write. A common estimate is that a programmer can write a program about ten times as fast in a high-level language as compared to assembly language.** That is just writing the program; it does not include problem definition, program design, debugging, testing, or documentation, all of which become simpler and faster. The high-level language program is, for instance, partly self-documenting. Even if you do not know FORTRAN, you probably could tell what the statement illustrated above does.

**MACHINE INDEPENDENCE OF HIGH-LEVEL LANGUAGES**

**High-level languages solve many other problems associated with assembly language programming.** The high-level language has its own syntax (usually defined by a national or international standard). The language does not mention the instruction set, registers, or other features of a particular computer. The compiler takes care of all such details. Programmers can concentrate on their own tasks; they do not need a detailed understanding of the underlying CPU architecture, for that matter, they do not need to know anything about the computer they are programming.

**PORTABILITY OF HIGH-LEVEL LANGUAGES**

**Programs written in a high-level language are portable — at least, in theory.** They will run on any computer or microcomputer that has a standard compiler for that language.

At the same time, all previous programs written in a high-level language for prior computers are available to you when programming a new computer. This can mean thousands of programs in the case of a common language like FORTRAN or BASIC.

## DISADVANTAGES OF HIGH-LEVEL LANGUAGES

**Well, if all the good things we have said about high-level languages are true, if you can write programs faster and make them portable besides, why bother with assembly languages? Who wants to worry about registers, instruction codes, mnemonics, and all that garbage! As usual there are disadvantages which balance the advantages.**

**SYNTAX OF HIGH-LEVEL LANGUAGES**

One obvious problem is that **you have to learn the "rules" or "syntax" of any high-level language** you want to use. A high-level language has a fairly complicated set of rules. You will find that it takes a lot of time just to get a program that is syntactically correct (and even then it probably will not do what you want). A high-level computer language is like a foreign language. If you have a little talent, you will get used to the rules and be able to turn out programs that the compiler will accept. Still, learning the rules and trying to get the program accepted by the compiler doesn't contribute directly to doing your job.

Here, for example, are some FORTRAN rules:

- Labels must be numbers in the first five card columns
- Statements must start in column seven
- Integer variables start with the letters I, J, K, L, M, or N

COST OF COMPILERS

Another obvious problem is that **you need a compiler to translate programs written in a high-level language.** Compilers are expensive and use a large amount of memory. While most assemblers occupy 2K to 16K bytes of memory (1K = 1024), compilers occupy 4K to 64K bytes. So the amount of overhead involved in using the compiler is rather large.

ALGEBRAIC NOTATION

Furthermore, **only some compilers will make the implementation of your task simpler.** FORTRAN, for example, is well-suited to problems that can be expressed as algebraic formulas. If, however, your problem is controlling a printer, editing a string of characters, or monitoring an alarm system, your problem cannot be easily expressed in algebraic notation. In fact, formulating the solution in algebraic notation may be more awkward and more difficult than formulating it in assembly language. The answer is, of course, to use a more suitable high-level language. Some such languages exist, but they are far less widely used and standardized than FORTRAN. You will not get many of the advantages of high-level languages if you use these so-called system implementation languages.

INEFFICIENCY OF HIGH-LEVEL LANGUAGES

**High-level languages do not produce very efficient machine language programs.** The basic reason for this is that compilation is an automatic process which is riddled with compromises to allow for many ranges of possibilities. The compiler works much like a computerized language translator — sometimes the words are right but the sounds and sentence structures are awkward. A simple compiler cannot know when a variable is no longer being used and can be discarded, or when a register should be used rather than a memory location, or when variables have simple relationships. The experienced programmer can take advantage of shortcuts to shorten execution time or reduce memory usage. A few compilers (known as optimizing compilers) can also do this but such compilers are much larger and slower than regular compilers.

The general advantages and disadvantages of high-level languages are:

**Advantages:**

ADVANTAGES OF HIGH-LEVEL LANGUAGES

- More convenient descriptions of tasks
- More efficient program writing
- Easier documentation
- Standard syntax
- Independence of the structure of a particular computer
- Portability
- Availability of library and other programs

**Disadvantages:**

DISADVANTAGES OF HIGH-LEVEL LANGUAGES

- Special rules
- Extensive hardware and software support required
- Orientation of common languages to algebraic or business problems
- Inefficient programs
- Difficulty of optimizing code to meet time and memory requirements
- Inability to use special features of a computer conveniently

## HIGH-LEVEL LANGUAGES FOR MICROPROCESSORS

**Microprocessor users will encounter several special difficulties when using high-level languages. Among these are:**

- **Few high-level languages exist for microprocessors**
- **No standard languages are widely available**
- **Few compilers actually run on microcomputers. Those that do often require very large amounts of memory**
- **Most microprocessor applications are not well-suited to high-level languages**
- **Memory costs are often critical in microprocessor applications**

The lack of high-level languages is partly a result of the fact that microprocessors are quite new and are the products of semiconductor manufacturers rather than computer manufacturers.

Very few high-level languages exist for microprocessors. The most common are the PL/I type languages such as Intel's PL/M, Motorola's MPL, and Signetics' PL$\mu$S.

Even the few high-level languages that exist do not conform to recognized standards, so the microprocessor user cannot expect to gain much program portability, access to program libraries, or use of previous experience or programs. The main advantages remaining are the reduction in programming effort and the smaller amount of detailed understanding of the computer architecture that is necessary.

**OVERHEAD FOR HIGH-LEVEL LANGUAGES**

The overhead involved in using a high-level language with microprocessors is considerable. Microprocessors themselves are better suited to control and slow interactive applications than they are to the character manipulation and language analysis involved in compilation. Therefore most compilers for microprocessors will not run on a microprocessor-based system. Instead, they require a much larger computer; i.e. they are cross-compilers rather than self-compilers. A user must not only bear the expense of the larger computer but must also physically transfer the program from the larger computer to the micro.

A few self-compilers are available. These compilers run on the microcomputer for which they produce object code. Unfortunately, they require large amounts of memory (16K or more), plus special supporting hardware and software.

**UNSUITABILITY OF HIGH-LEVEL LANGUAGES**

High-level languages also are not generally well-suited to microprocessor applications. Most of the common languages were devised either to help solve scientific problems or to handle large-scale business data processing. Few microprocessor applications fall in either of these areas. Most microprocessor applications involve sending data and control information to output devices, and receiving data and status information from input devices. Often the control and status information consists of a few binary digits with very precise hardware-related meanings. If you try to write a typical control program in a high-level language, you often feel like someone who is trying to eat soup with chopsticks. For tasks in such areas as test equipment, terminals, navigation systems, and business equipment, the high-level languages work much better than they do in instrumentation, communications, peripherals, and automotive applications.

**APPLICATION AREAS FOR LANGUAGE LEVELS**

Applications better suited to high-level languages are those which require large memories. If, as in a valve controller, electronic game, appliance controller, or small instrument, the cost of a single memory chip is important, then the inefficiency of high-level languages is intolerable. If, on the other hand, as in a terminal or test equipment, the system has many thousands of bytes of memory anyway, the inefficiency of high-level languages is not as important. Clearly the size of the program and

the volume of the product are important factors as well. A large program will greatly increase the advantages of high-level languages. On the other hand, a high-volume application will mean that fixed software development costs are not as important as memory costs which are part of each system.

## WHICH LEVEL SHOULD YOU USE?

**That depends on your particular application. Let us briefly note some of the factors which may favor particular levels:**

**Machine Language:**

**APPLICATIONS FOR MACHINE LANGUAGE**

- **Low-volume applications involving small, simple programs**
- **Applications where the prototype is the final product**
- **Simple control applications involving a limited number of calculations**

**Assembly Language:**

**APPLICATIONS FOR ASSEMBLY LANGUAGE**

- **Small to moderate sized programs**
- **Applications where memory cost is a factor**
- **Real-time control applications**
- **Limited data processing**
- **High-volume applications**

**High-Level Languages:**

**APPLICATIONS FOR HIGH-LEVEL LANGUAGE**

- **Large programs**
- **Low-volume applications requiring long programs**
- **Applications requiring large memories**
- **More computation than input/output or control**
- **Compatibility with similar applications using larger computers**
- **Availability of specific programs in a high-level language which can be used in the application**

Many other factors are also important, such as the availability of a larger computer for use in development, experience with particular languages, and compatibility with other applications.

If hardware will ultimately be the largest cost in your application, you should favor assembly language. But be prepared to spend extra time in software development in exchange for lower memory costs and higher execution speeds. If software will be the largest cost in your application, you should favor a high-level language. But be prepared to spend the extra money required for the supporting hardware and software.

Of course, no one except some theorists will object if you use both assembly and high-level languages. You can write the program originally in a high-level language and then patch some sections in assembly language. However, most users prefer not to do this because of the havoc it creates in debugging, testing, and documentation.

## HOW ABOUT THE FUTURE?

**We expect that the future will tend to favor high-level languages** for the following reasons:

**FUTURE TRENDS IN LANGUAGE LEVELS**

- Programs always seem to add extra features and grow larger
- Hardware and memory are becoming less expensive
- Software and programmers are becoming more expensive
- Memory chips are becoming available in larger sizes, at lower "per bit" cost, so actual savings in chips are less likely
- More compilers are becoming available
- More suitable and more efficient high-level languages are being developed
- More standardization of high-level languages will occur

Assembly language programming of microprocessors will not be a dying art any more than it is now for large computers. But longer programs, cheaper memory, and more expensive programmers will make software costs a larger part of most applications. The edge in many applications will therefore go to high-level languages.

## WHY THIS BOOK?

**If the future would seem to favor high-level languages, why have a book on assembly language programming? The reasons are:**

1) Most current microcomputer users program in assembly language (almost 2/3, according to one recent survey).
2) Many microcomputer users will continue to program in assembly language since they need the detailed control that it provides.
3) No suitable high-level language has yet become widely available or standardized.
4) Many applications require the efficiency of assembly language.
5) An understanding of assembly language can help in evaluating high-level languages.

The rest of this book will deal exclusively with assemblers and assembly language programming. However, we do want readers to know that assembly language is not the only alternative. You should watch for new developments that may significantly reduce programming costs if such costs are a major factor in your application.

# Chapter 2
# ASSEMBLERS

This chapter discusses the functions performed by assemblers, beginning with features common to most assemblers, and proceeding through more elaborate capabilities such as macros and conditional assembly. You may wish to skim this chapter for the present and return to it when you feel more comfortable with the material.

## FEATURES OF ASSEMBLERS

**As we mentioned previously, today's assemblers do much more than translate assembly language mnemonics into binary codes. But we will first describe how an assembler handles the translation of mnemonics before describing additional assembler features. Finally we will explain how assemblers are used.**

### ASSEMBLER INSTRUCTIONS

**ASSEMBLY LANGUAGE FIELDS**

**Assembly language instructions (or "statements") are divided into a number of fields, as shown in Table 2-1.**

**The operation code field is the only field which can never be empty; it always contains either an instruction mnemonic or a directive to the assembler, called a pseudo-instruction, pseudo-operation, or pseudo-op.**

**The address field may contain an address or data, or it may be blank.**

Table 2-1. The Fields Of An Assembly Language Instruction

| Label Field | Operation Code Or Mnemonic Field | Operand Or Address Field | Comment Field |
|---|---|---|---|
| START | LDA | VAL1 | LOAD FIRST NUMBER INTO A |
| | MOV | B,A | SAVE IN B |
| | LDA | VAL2 | LOAD SECOND NUMBER INTO A |
| | ADD | B | ADD FIRST NUMBER TO A |
| | STA | SUM | STORE SUM |
| NEXT | ? | ? | NEXT INSTRUCTION |
| VAL1 | DS | | |
| VAL2 | DS | | |
| SUM | DS | | |

**The comment and label fields are optional. A programmer will assign a label to a statement or add a comment as a personal convenience, e.g., to make the program easier to read.**

FORMAT

**Of course, the assembler must have some way of telling where one field ends and another begins.** Assemblers which use punched card input often require that each field start in a specific card column. This is a fixed format. However, fixed formats may be inconvenient when the input medium is paper tape; fixed formats are also a nuisance to programmers. The alternative is a free format where the fields may appear anywhere on the line.

DELIMITERS

If the assembler cannot use the location on the line to tell the fields apart, it must use something else. **Most assemblers use a special symbol or delimiter at the beginning or end of each field.** The most obvious delimiter is the space character. Commas, periods, semi-colons, colons, slashes, question marks and other characters which would not otherwise be used in assembly language programs also may serve as delimiters. Table 2-2 lists standard Intel 8080 assembler delimiters.

Table 2-2. Standard 8080 Assembler Delimiters

| | |
|---|---|
| : | after a label |
| 'space' | between operation code and address |
| , | between operands in the address field |
| ; | before a comment |

**You will have to exercise a little care with delimiters. Some assemblers are fussy about extra spaces or the appearance of delimiters in comments or labels. A well-written assembler will handle these minor problems, but many assemblers are not well-written. Our recommendation is simple: avoid potential problems if you can. The following rules will help:**

1) Do not use extra spaces, particularly after commas which separate operands.
2) Do not use delimiter characters in names or labels.
3) Include standard delimiters even if your assembler does not require them. Your programs will then be assembled by any assembler.

## LABELS

LABEL FIELD

**The label field is the first field in an assembly language instruction;** it may be blank. If a label is present, the assembler assigns to the label the value of the address for the memory location into which the first object program byte for that instruction is loaded. You may subsequently use the label as an address or as data in another instruction's operand field. The assembler will replace the label with the assigned value when creating an object program.

LABELS IN JUMP INSTRUCTIONS

**Labels are most frequently used in Jump, Call or Branch instructions.** These instructions place a new value in the Program Counter and so alter the normal sequential execution of instructions. JUMP $150_{16}$ means "place the value $150_{16}$ in the Program Counter". The next instruction to be executed will be the one in memory location $150_{16}$. The instruction JUMP START means "place the value assigned to the label START in the Program Counter". The next instruction to be executed will be the one in the memory location to which the label START has been assigned. Table 2-3 contains an example.

Table 2-3. Assigning And Using A Label

```
ASSEMBLY LANGUAGE PROGRAM

START     LOAD ACCUMULATOR 100
          .
          .
          . MAIN PROGRAM
          .
          .
          .
          JUMP START
```

When the machine language version of this program is executed, the instruction JUMP START causes the address of the instruction labeled START to be placed in the Program Counter. The instruction with the label START will be executed next.

**Why use a label? Here are some reasons:**

1) A label makes a program location easier to find and remember.
2) The label can be moved to correct a program. You do not have to change any subsequent label references; the assembler will make all the necessary changes.
3) The assembler or loader can relocate the whole program by adding a constant (a relocation constant) to each address in which a label was used. Thus we can move the program to allow for the insertion of other programs or simply to rearrange memory.
4) The program is easier to use as a library program, i.e., it is easier for someone else to take your program and add it to some totally different program.
5) You do not have to figure out memory addresses. Figuring out memory addresses is particularly difficult with microprocessors which have instructions that vary in length.

It makes sense to assign a label to any instruction which you might want to use as a destination or otherwise identify.

**CHOOSING LABELS**

**The next question is what label to use. The assembler often places some restrictions** on the number of characters (usually 5 or 6), the leading character (often must be a letter), and the trailing characters (often must be letters, numbers, or one of a few special characters). Beyond these restrictions, the choice is up to you.

Our own preference is to use labels that suggest their purpose, i.e., mnemonic labels. Typical examples are ADDW in a routine that adds one word into a sum, SRETX in a routine that searches for the ASCII character ETX, or NKEYS for a location in data memory that contains the number of key entries. Meaningful labels are easier to remember and contribute to program documentation. Some programmers prefer to use a standard label format, such as starting with L0000. These labels are self-sequencing (you can skip a few numbers to permit insertions), but they do not help document the program.

**Some label selection rules will keep you out of trouble. We recommend the following:**

**RULES OF LABELING**

1) Do not use labels which are the same as operation codes or other mnemonics. Most assemblers will not allow this usage; others will, but it is very confusing.
2) Do not use labels which are longer than the assembler permits. Assemblers have various truncation rules.
3) Avoid special characters (non-alphabetic and non-numeric). Some assemblers will not permit them; others only allow certain ones. The simplest practice is to stick to letters and numbers.
4) Start each label with a letter. Such labels are always acceptable.
5) Do not use labels that could be confused with each other. Avoid the letters I, O and Z and the numbers 0, 1 and 2. Also avoid things like XXXX and XXXXX. There's no sense in tempting fate and Murphy's laws.
6) When you are not sure if a label is legal, do not use it. You will not get any real benefit from discovering exactly what the assembler will accept.

These are recommendations, not rules. You do not have to follow them but don't blame us if you waste time on silly problems.

## ASSEMBLER OPERATION CODES (MNEMONICS)

**The main task of the assembler is the translation of mnemonic operation codes into their binary equivalents.** The assembler performs this task using a fixed table much as you would if you were doing the assembly by hand.

**The assembler** must, however, do more than just translate the operation codes. It **must also somehow determine how many operands the instruction requires and what type they are.** This may be rather complex — some instructions (like a Halt) have no operands, others (like an Addition or a Jump instruction) have one, while still others (like a transfer between registers or a multiple-bit shift) require two. Some instructions may even allow alternatives, e.g., some computers have instructions (like Shift or Clear) which can either apply to the Accumulator or to a memory location. We will not discuss how the assembler makes these distinctions; we will just note that it must do so.

## PSEUDO-OPERATIONS

**ASSEMBLER DIRECTIVE**

**Some assembly language instructions are not directly translated into machine language instructions. These instructions are directives to the assembler;** they assign the program to certain areas in memory, define symbols, designate areas of RAM for temporary data storage, place tables or other fixed data in memory, and perform other housekeeping functions.

To use these directives or pseudo-operations a programmer places the pseudo-operation's mnemonic in the operation code field, and an address or data in the address field, if the specified pseudo-operation requires it.

The most common pseudo-operations are:

DATA
EQUATE or DEFINE
ORIGIN
RESERVE

Different assemblers use different names for these operations but the purposes are the same. Housekeeping pseudo-operations include:

END
LIST
NAME
PAGE
SPACE
TITLE

We will discuss these pseudo-operations briefly, although their functions are usually obvious.

## THE DATA PSEUDO-OPERATION

**The DATA pseudo-operation allows the programmer to enter data into program memory.** This data may include:

- Lookup tables
- Code conversion tables
- Messages
- Synchronization patterns
- Thresholds
- Names
- Coefficients for equations
- Commands
- Conversion factors
- Weighting factors
- Characteristic times or frequencies
- Subroutine addresses
- Key identifications
- Test patterns
- Character generation patterns
- Identification patterns
- Tax tables
- Standard forms
- Masking patterns

The DATA pseudo-operation treats the data as a permanent part of the program.

**The format of a DATA pseudo-operation is usually quite simple. An instruction like:**

```
DZCON   DATA    12
```

**will place the number 12 in the next available memory location and assign that location the name DZCON.** Usually every DATA pseudo-operation has a label, unless it is one of a series of DATA pseudo-operations. The data and label may take any form that the assembler permits.

Most assemblers allow more elaborate DATA instructions which handle a large amount of data at one time, e.g.,

```
EMESS   DATA    'ERROR'
SQRS    DATA    1,4,9,16,25
```

A single instruction may fill many words of program memory, limited only by the length of a line. Note that if you cannot get all the data on one line, you can always follow one

DATA instruction with another, e.g.,

```
MESSG   DATA    'NOW IS THE'
        DATA    'TIME FOR ALL'
        DATA    'GOOD MEN'
        DATA    'TO COME TO THE'
        DATA    'AID OF THEIR'
        DATA    'COUNTRY'
```

**Microprocessor assemblers typically have some variations of standard DATA pseudo-operations.** DEFINE BYTE OR FORM CONSTANT BYTE handles 8-bit numbers; DEFINE WORD OR FORM CONSTANT WORD handles 16-bit numbers or addresses. Other special pseudo-operations may handle character-coded data.

## THE EQUATE (or EQUALS) PSEUDO-OPERATION

DEFINING NAMES

**The EQUATE pseudo-operation allows the programmer to equate labels and names with addresses or data. This pseudo-operation is almost always given the mnemonic EQU.** The names may refer to device addresses, numeric data, starting addresses, fixed addresses, etc.

**The EQUATE pseudo-operation assigns the numeric value in its operand field to the label in its label field.** Here are two examples:

```
TTY     EQU     5
LAST    EQU     5000
```

Most assemblers will allow you to define one label in terms of another, e.g.,

```
LAST    EQU     FINAL
ST1     EQU     START+1
```

The label in the operand field must, of course, have been previously defined. Often, the operand field may contain more complex expressions, as we shall see later. Double name assignments (two names for the same data or address) may be useful in patching together programs which use different names for the same variable (or different spellings of what was supposed to be the same name).

SYMBOL TABLE

**Note that an EQU pseudo-operation does not result in the assembler placing anything into memory. It simply places an additional name in a table (called a symbol table) which the assembler maintains.** This table, unlike the mnemonic table, must be in RAM since it varies with each program. The assembler program will always need some RAM to hold the symbol table; the more RAM it has, the more symbols it can accept. This RAM is in addition to any which the assembler needs as temporary storage.

USE OF NAMES

When do you use a name? The answer is whenever you have a parameter that has some meaning besides its ordinary numeric value, or the numeric value of the parameter might be changed. We typically assign names to time constants, device addresses, masking patterns, conversion factors, and the like. A name like DELAY, TTY, KBD, NROW, or OPEN not only makes the parameter easier to change, but it also adds to program documentation. We also assign names to memory locations that have special purposes; they may hold data, mark the start of the program, or be available for intermediate storage.

CHOICE OF NAMES

**What name do you use?** The best rules are much the same as in the case of labels, except that here meaningful names really count. Why not call the teletypewriter TTY instead of X15, a bit time delay BTIME or BTDLY rather than WW, the number of the "GO" key on a keyboard GOKEY rather than HORSE? This advice seems straightforward but a surprising number of programmers do not follow it.

**Where do you place the EQUATE pseudo-operations? The best place is at the start of the program,** under appropriate comment headings such as I/O ADDRESSES, TEMPORARY STORAGE, TIME CONSTANTS, or PROGRAM LOCATIONS. This makes the definitions easy to find if you want to change them. Furthermore, another user will be able to look up all the definitions in one centralized place. Clearly this practice improves documentation and makes the program easier to use.

**PLACEMENT OF DEFINITIONS**

Definitions used only in a specific subroutine should appear at the start of the subroutine.

## THE ORIGIN PSEUDO-OPERATION

**The ORIGIN pseudo-operation (almost always abbreviated ORG) allows the programmer to locate programs, subroutines, or data anywhere in memory.** Programs and data may be located in different areas of memory depending on the memory configuration. Startup routines, interrupt service routines, and other required programs may be scattered around memory at suitable addresses.

**The assembler maintains a Location Counter (comparable to the computer's Program Counter) which contains the location in memory of the next instruction or data item being processed.** An ORG pseudo-operation causes the assembler to place a new value in the Location Counter, much as a Jump instruction causes the CPU to place a new value in the Program Counter. The output from the assembler must not only contain instructions and data, but must also indicate to the loader program where in memory it should place the instructions and data.

**LOCATION COUNTER**

Microprocessor programs often contain several ORIGIN statements for the following purposes:

Reset (startup) address
Interrupt service addresses
Trap addresses
RAM storage
Memory stack

Still other ORIGIN statements may allow room for later insertions, place tables or data in memory, or assign vacant RAM space for data buffers. Program and data memory in microcomputers may occupy widely scattered addresses.

**Typical ORIGIN statements are:**

```
ORG     RESET
ORG     1000
ORG     INT3
```

Some assemblers assume an origin of zero if the programmer does not put an ORG statement at the start of the program. The convenience is slight; we recommend the inclusion of an ORG statement to avoid confusion.

## THE RESERVE PSEUDO-OPERATION

**The RESERVE pseudo-operation allows the programmer to allocate RAM for various purposes such as data tables, temporary storage, indirect addresses, a Stack, etc.**

**ALLOCATING RAM**

**Using the RESERVE pseudo-operation, you assign a name to the memory area and declare the number of locations to be assigned.** Here are some examples:

```
NOKEY   RESERVE   1
TEMP    RESERVE   50
VOLTG   RESERVE   80
BUFR    RESERVE   100
```

You can use the RESERVE pseudo-operation to reserve memory locations in program memory or in data memory; however the nature of the RESERVE pseudo-operation is more meaningful when applied to data memory.

In reality, all the RESERVE pseudo-operation does is increase the assembler's Location Counter by the amount declared in the operand field. The assembler does not actually produce any object code at all.

**Note the following features of RESERVE:**

1) The label of RESERVE pseudo-operation is assigned the value of the first address reserved. For example, the sequence:

```
        ORG       3000
BUF1    RESERVE   100
BUF2    RESERVE   50
VOLTS   RESERVE   5
```

   assigns to the label BUF1 the value 3000, to BUF2 3100, and to VOLTS 3150.

2) You must specify the number of locations to be reserved. There is no default case.

3) No data is placed in the reserved locations. Any data that, by chance, may be in these locations will be left there.

**INITIALIZING RAM**

**Some assemblers allow the programmer to place initial values in RAM. We strongly recommend that you do not use this feature** — it assumes that the program (along with the initial values) will be loaded from an external device (e.g., paper tape or floppy disk) each time it is run. Most microprocessor programs, on the other hand, reside in non-volatile ROM and start when power comes on. The RAM in such situations does not retain its contents, nor is it reloaded. Always include instructions to initialize the RAM in your program.

## HOUSEKEEPING PSEUDO-OPERATIONS

**There are various housekeeping pseudo-operations which affect the operation of the assembler and its program listing rather than the output program itself. Common housekeeping pseudo-operations include:**

1) END, which marks the end of the assembly language source program.

2) LIST, which tells the assembler to print the source program. Some assemblers allow such variations as NO LIST or LIST SYMBOL TABLE to avoid long, repetitive listings.

3) NAME or TITLE, which prints a name at the top of each page of the listing.

4) PAGE or SPACE, which skips to the next page or next line, respectively, and improves the appearance of the listing, making it easier to read.

5) PUNCH, which transfers subsequent object code to the paper tape punch. This pseudo-operation may in some cases be the default option, and therefore unnecessary.

## LABELS WITH PSEUDO-OPERATIONS

**Users often wonder if or when they can assign a label to a pseudo-operation. These are our recommendations:**

1) All EQUATE pseudo-operations must have labels; they do not make any sense otherwise.
2) DATA and RESERVE pseudo-operations usually have labels. The label identifies the first memory location used or assigned.
3) Other pseudo-operations should not have labels. Some assemblers allow other pseudo-operations to have labels, but the meaning of the labels varies. We recommend that you avoid this practice.

# ADDRESSES AND THE OPERAND FIELD

**Most assemblers allow the programmer a lot of freedom in describing the contents of the Operand Address field. But remember, the assembler has built-in names for registers and instructions and may have other built-in names.**

**Some common options for the operand field are:**

DECIMAL DATA OR ADDRESSES

**1) Decimal numbers**

Most assemblers assume all numbers to be decimal unless they are marked otherwise. So

```
ADD     100
```

means "add the contents of memory location 100 decimal to the contents of the Accumulator".

OTHER NUMBER SYSTEMS

**2) Other number systems**

Most assemblers will also accept binary, octal, or hexadecimal entries. But you must identify these number systems in some way, e.g., by preceding or following the number with an identifying character or letter. Here are some common identifiers:

B for binary

O, Q, or C for octal (we avoid O because of the confusion with zero).

H for hexadecimal (or standard BCD)

D for decimal. D may be omitted, it is the default case.

Assemblers generally require hexadecimal numbers to start with a digit (e.g., 0A36 instead of A36) in order to distinguish between numbers and names or labels. It is good practice to enter numbers in the base in which their meaning is the clearest — i.e., decimal constants in decimal; addresses and BCD numbers in hexadecimal; masking patterns or bit outputs in binary if they are short, and in hexadecimal if they are long.

**3) Symbolic names**

Names can appear in the operand field; they will be treated as the data which they represent. But remember,there is a difference between data and addresses. The sequence

```
FIVE    EQU     5
        ADD     FIVE
```

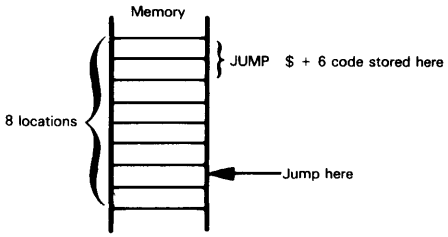
will add the contents of memory location 5 (not necessarily the number 5) to the contents of the Accumulator.

**4) The current value of the location counter (usually referred to as * or $).**

This is useful mainly in Jump instructions; for example:

```
JUMP    $+6
```

causes a Jump to the memory location six words beyond the word which contains the first byte of the JUMP instruction:




Most microprocessors have many two and three-word instructions. Thus you will have difficulty determining exactly how far apart two assembly language statements are. Therefore, using offsets from the Location Counter frequently results in errors which you can avoid if you use labels.

**5) Character codes**

**ASCII CHARACTERS**

Most assemblers allow text to be entered as ASCII strings. Such strings may be surrounded either with single or double quotation marks; strings may also use a beginning or ending symbol such as A or C. A few assemblers also permit EBCDIC strings.

We recommend that you use character strings for all text. It improves the clarity and readability of the program.

**6) Combination of 1) through 5) with arithmetic, logical, or special operators.**

**ARITHMETIC AND LOGICAL EXPRESSIONS**

Almost all assemblers allow simple arithmetic combinations such as START+1. Some assemblers also permit multiplication, division, logical functions, shifts, etc. These are referred to as expressions. Note that the assembler evaluates expressions at assembly time. Even though an expression in the operand field may involve multiplication, you may not be able to use multiplication in the logic of your own program — unless you write a subroutine for that specific purpose.

Assemblers vary in what expressions they accept and how they interpret them. Complex expressions make a program difficult to read and understand.

We have made some recommendations during this section but will repeat them and add others here. In general, the user should emphasize clarity and simplicity. There is no payoff for being an expert in the intricacies of the assemblers or in having the most complex expression on the block. We suggest the following approach:

1) Use the clearest number system or character code for data.
2) Masks and BCD numbers in decimal, ASCII characters in octal, or ordinary numerical constants in hexadecimal serve no purpose and therefore should not be used.
3) Remember to distinguish data and addresses.
4) Don't use offsets from the Location Counter.
5) Keep expressions simple and obvious. Don't rely on obscure features of the assembler.

# CONDITIONAL ASSEMBLY

**Some assemblers allow you to include or exclude parts of the source program, depending on conditions existing at assembly time. This is called conditional assembly;** it gives the assembler some of the flexibility of a compiler. **Most microcomputer assemblers have limited capabilities for conditional assembly. A usual form is:**

```
IF COND
.
.CONDITIONAL PROGRAM
.
.
ENDIF
```

If the expression COND is true at assembly time, the instructions between IF and ENDIF (two pseudo-operations) are included in the program.

**Typical uses of conditional assembly are:**

1) To include or exclude extra variables.
2) To place diagnostics in test runs.
3) To allow data of various bit lengths.

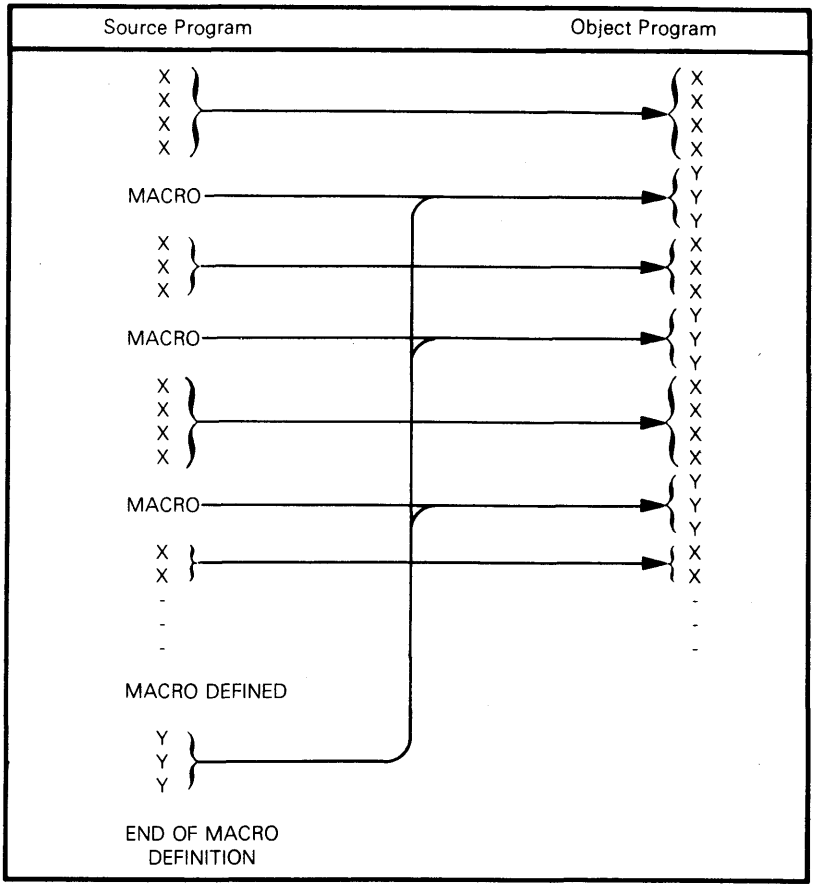
Unfortunately, conditional assembly tends to clutter programs and make them difficult to read. Use conditional assembly only if it is necessary.

# MACROS

**DEFINING A SEQUENCE OF INSTRUCTIONS**

You will often find that particular sequences of instructions occur many times in a source program. Repeated instruction sequences may reflect the needs of your program logic, or they may be compensating for deficiencies in your microprocessor's instruction set. You can avoid repeatedly writing out the same instruction sequence by using a macro.

**Macros allow you to assign a name to an instruction sequence. You then use the macro name in your source program instead of the repeated instruction sequence. The assembler will replace the macro name with the appropriate sequence of instructions.** This may be illustrated as follows:



Macros are not the same as subroutines. A subroutine occurs once in a program, and program execution branches to the subroutine. A macro is expanded to an actual instruction sequence each time the macro occurs; thus a macro does not cause any branching.

**ADVANTAGES OF MACROS**

**Macros have the following advantages:**

1) Shorter source programs.
2) Better program documentation.
3) Use of debugged instruction sequences — once the macro has been debugged, you are sure of an error free instruction sequence every time you use the macro.
4) Easier changes. Change the macro definition and the assembler makes the change for you every time the macro is used.
5) Inclusion of commands, keywords, or other computer instructions in the basic instruction set. You use the macro as an extension of your instruction set.

**DISADVANTAGES OF MACROS**

The disadvantages of macros are:

1) Repetition of the same instruction sequences.
2) A single macro may create a lot of instructions.
3) Lack of standardization.
4) Possible effects on registers and flags that may not be clearly stated.

**LOCAL OR GLOBAL VARIABLES**

One problem is that variables used in a macro are only known within it (i.e., they are local rather than global). This can often create a great deal of confusion without any gain in return. You should be aware of this problem when using macros.

# COMMENTS

**All assemblers allow you to place comments in a source program. Comments have no effect on the object code, but they help you to read, understand, and document the program. Good commenting is an essential part of writing assembly language programs; without comments programs are very difficult to understand.**

**COMMENTING TECHNIQUES**

**We will discuss commenting along with documentation in a later chapter, but here are some guidelines:**

1) Use comments to tell what the program is doing, *not what instructions do.*

   Comments should say things like "IS TEMPERATURE ABOVE LIMIT?", "LINE FEED TO TTY", OR "EXAMINE LOAD SWITCH".

   Comments should not say things like "ADD 1 TO ACCUMULATOR", "JUMP TO START", or "LOOK AT CARRY". You should describe how the program is affecting the system; internal effects on the CPU are seldom of any interest.

2) Keep comments brief and to the point. Details should be available elsewhere in the documentation.
3) Comment all key points.
4) Do not comment standard instructions or sequences which change counters and pointers; pay special attention to instructions that may not have an obvious meaning.
5) Do not use obscure abbreviations.
6) Make the comments neat and readable.
7) Comment all definitions, describing their purposes. Also mark all tables and data storage areas.

8) Comment sections of the program as well as individual instructions.

9) Be consistent in your terminology. You can (should) be repetitive; you do not need to consult a thesaurus.

10) Leave yourself notes at points which you find confusing, e.g., "REMEMBER CARRY WAS SET BY LAST INSTRUCTION". You may drop these in the final documentation.

A well-commented program is easy to work with. You will recover the time spent in commenting many times over. We will try to show good commenting style in the programming examples, although we often over-comment for instructional purposes.

## TYPES OF ASSEMBLERS

**Although all assemblers perform the same tasks, their implementations vary greatly. We will not try to describe all the existing types of assemblers; we will merely define the terms and indicate some of the choices.**

**CROSS-ASSEMBLER**

**A cross-assembler is an assembler that runs on a computer other than the one for which it assembles object programs.**

The computer on which the cross-assembler runs is typically a large computer with extensive software support and fast peripherals — such as an IBM 360 or 370, a Univac 1108, or a Burroughs 6700. The computer for which the cross-assembler assembles programs is typically a micro like the Intel 8080 or Motorola 6800. Most cross-assemblers are written in FORTRAN so that they are portable.

**RESIDENT ASSEMBLER**

A self-assembler or resident assembler is an assembler that runs on the computer for which it assembles programs. The self-assembler will require some memory and peripherals, and it may run quite slowly.

**MACRO-ASSEMBLER**

**A macroassembler is an assembler that allows you to define sequences of instructions as macros.**

**MICRO-ASSEMBLER**

**A microassembler is an assembler used to write the microprograms which define the instruction set of a computer. Microprogramming has nothing specifically to do with microcomputers. Microprogramming is described conceptually in "An Introduction To Microcomputers: Volume I — Basic Concepts", Chapter 4.**

**META-ASSEMBLER**

**A meta-assembler is an assembler which can handle many different instruction sets.** The user must define the particular instruction set being used.

**ONE-PASS ASSEMBLER**

**A one-pass assembler is an assembler which only goes through the assembly language program once.** Such an assembler must have some way of resolving forward references, e.g. Jump instructions which use labels that appear later in the source program, i.e., that have not yet been defined.

**TWO-PASS ASSEMBLER**

**A two-pass assembler is an assembler which goes through the assembly language source program twice. The first time the assembler simply collects and defines all the symbols; the second time it replaces the references with the actual definitions. A two-pass assembler has no problems with forward references but may be quite slow if no backup storage (like a floppy disk) is available; then the assembler must physically read the program twice from a slow input medium (like a teletypewriter paper tape reader). Most microprocessor-based assemblers require two passes.**

# ERRORS

**Assemblers normally provide error messages, often consisting of a single coded letter. Some typical errors are:**

1) Undefined name (often a misspelling or an omitted definition).
2) Illegal character (e.g., a 2 in a binary number).
3) Illegal format (wrong delimiter or incorrect operands).
4) Invalid expression (e.g., two operators in a row).
5) Illegal value (usually too large).
6) Missing operand.
7) Double definition (i.e., two different values assigned to one name).
8) Illegal label (e.g., a label on a pseudo-operation that cannot have one).
9) Missing label.
10) Undefined operation code.

In interpreting assembler errors, you must remember that the assembler may get off on the wrong track if it finds a stray letter, an extra space, or incorrect punctuation. Many assemblers will then proceed to misinterpret several succeeding statements and produce meaningless error messages. Always look at the first error very carefully; subsequent ones may depend on it. Caution and consistent adherence to standard formats will eliminate many annoying mistakes.

# LOADERS

The loader is the program which actually takes the output (object code) from the assembler and places it in memory. Loaders range from the very simple to the very complex. We will describe a few different types.

**BOOTSTRAP LOADER**

**A bootstrap loader is a program which uses its own first few instructions to load the rest of itself, or another loader program into memory.** The bootstrap loader may be in ROM, or you may have to enter it into the computer memory using front panel switches. The assembler may place a bootstrap loader at the start of the object program which it produces.

**RELOCATING LOADER**

**A relocating loader can load programs anywhere in memory.** It typically loads each program into the memory space immediately following that used by the previous program. The programs, however, must themselves be capable of being moved around in this way, i.e., they must be relocatable. An absolute loader, in contrast, will always place the programs in the same area of memory.

**LINKING LOADERS**

**A linking loader loads programs and subroutines as separate modules; it resolves cross-references** — that is, an instruction in one module which refers to a label in another module. Object programs loaded by a linking loader must be created by an assembler which permits and marks cross-references.

# Chapter 3
# THE 8080A AND 8085 ASSEMBLY LANGUAGE INSTRUCTION SETS

**We are now ready to start creating assembly language programs. We begin in this chapter by defining the individual instructions of the 8080A and 8085 assembly language instruction sets, plus the syntax rules of the Intel assemblers.**

Instruction sets for the 8080A and 8085 microprocessors are identical apart from two additional instructions (RIM and SIM), available only with the 8085. There are also some differences in instruction execution cycles. Table 3-5 identifies these differences.

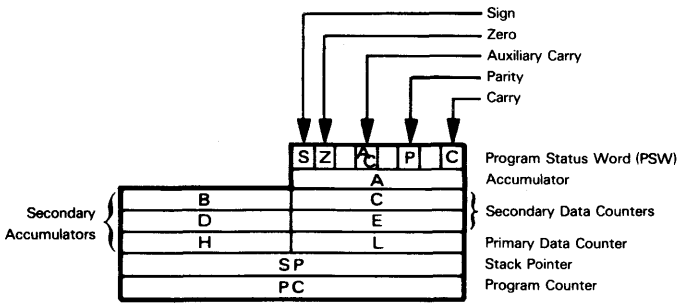
**We do not discuss any aspects of microcomputer hardware,** signals, interfaces, or CPU architecture in this book. This information is described in detail in An Introduction To Microcomputers: Volume II — Some Real Products while 8080 Programming For Logic Design discusses assembly language as an extension of digital logic. In this book, **we look at programming techniques from the assembly language programmer's viewpoint, where pins and signals are irrelevant and there are no important differences between a minicomputer and a microcomputer.**

Interrupts. Direct Memory Access and Stack architectures for the 8080A and 8085 will be described in later chapters of this book, in conjunction with assembly language programming discussions of the same subjects.

The definitions of the instruction set given in this chapter consist of a detailed definition of each assembly language instruction. These definitions are identical to those found in Chapter 6 of 8080 Programming For Logic Design , except for the two new 8085 instructions (RIM and SIM). The detailed description of individual instructions is preceded by a general discussion of the 8080A and 8085 instruction set that divides instructions into those which are commonly used, infrequently used and rarely used. If you are an experienced assembly language programmer, this categorization is not particularly important — and depending on your own programming prejudices, it may not even be accurate. If you are a novice assembly language programmer, we recommend that you begin by writing programs using only instructions in the "commonly used" category. Once you have mastered the concepts of assembly language programming, you may examine other instructions and use them where appropriate.

## CPU REGISTERS AND STATUS FLAGS

**The CPU registers and status flags are identical for the 8080A and 8085 CPUs. The registers and status flags may be illustrated as follows:**




**The Accumulator is the primary source and destination for one operand and two operand instructions.** For example, all data transfers between the CPU and I/O devices are performed through the Accumulator. In addition, many more memory reference instructions move data between the Accumulator and memory than between any other register and memory. All arithmetic and Boolean instructions take one of the operands from the Accumulator and return the result to the Accumulator. An instruction must therefore **load the Accumulator before the 8080 can perform any arithmetic or Boolean operations.**

**The B, C, D, E, H and L registers are all secondary registers.** Data stored in any of these six registers may be accessed with equal ease; such data can be moved to any other register or can be used as the second operand in two operand instructions.

There are, however, some important differences in the functions of Registers B, C, D, E, H and L.

**Registers H and L are the primary Data Pointer for the 8080A and 8085.** That is to say, you will normally use these two registers to hold the 16-bit memory address of data being accessed. Data may be transferred between any registers and the memory location addressed by H and L. Using any other method of addressing memory, only the Accumulator can act as the source or destination of data within the CPU. Therefore the 8080A or 8085 programmer should try to address data memory via Registers H and L whenever possible.

**Within your program logic always reserve Registers H and L to hold a data memory address.**

**Registers B, C, D and E provide secondary data storage;** frequently the second operand for two operand instructions is stored in one of these four registers. (The first operand is stored in the Accumulator, which also is the destination for the result).

There are **a limited number of instructions that treat Registers B and C, or D and E as 16-bit Data Pointers.** But these instructions move data between memory and the Accumulator only.

**In your program logic you should normally use Registers B, C, D and E as temporary storage for data or addresses.**

**There are a number of instructions which handle 16 bits of data at a time. These instructions refer to pairs of CPU registers as follows:**

| | | |
|---|---|---|
| PSW | and | A |
| B | and | C |
| D | and | E |
| H | and | L |
| High order byte | | Low order byte |

The combination of the Accumulator and Program Status Word, treated as a 16-bit unit, is used only for Stack operations. Arithmetic operations access B and C or D and E or H and L as 16-bit data units.

**The Carry Status flag holds Carries out of the most significant bit in any arithmetic operation.** The Carry flag is also included in Shift instructions; it is reset by Boolean instructions.

**The Parity status flag is set to 1 whenever an arithmetic or Boolean operation produces a result which contains an even number of 1 bits. It is cleared whenever such an operation produces a result with an odd number of 1 bits.**

**The Zero flag is set to 1 when any arithmetic or Boolean operation generates a 0 result. The Zero status is set to 0 when such an operation generates a non-zero result.**

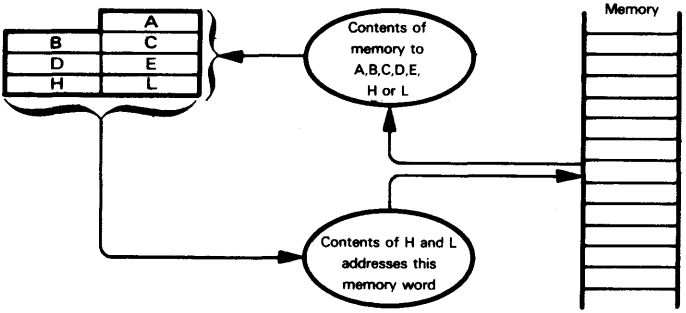
**The Sign status flag acquires the value of the most significant bit of the result following the execution of any arithmetic or Boolean instruction.**

**The Auxiliary Carry status flag holds any Carry from bit 3 to 4 resulting from the execution of an arithmetic instruction.** The purpose of this status flag is to simplify Binary-Coded-Decimal (BCD) operations; this is the standard use of an Auxiliary Carry status flag as described in An Introduction To Microcomputers: Volume I , Chapter 3.

**The 16-bit Stack Pointer allows you to implement a stack anywhere in addressable memory.** The size of the stack is limited only by the amount of addressable memory present. In reality you will rarely use more than 256 bytes of memory for your stack. You should use the stack for accessing subroutines and processing interrupts. Do not use the stack to pass parameters to subroutines. This is not very efficient within the limitations of the 8080A instruction set. The 8080A/8085 stack is started at its highest address. A Push decrements the Stack Pointer contents; a Pop increments the Stack Pointer contents.

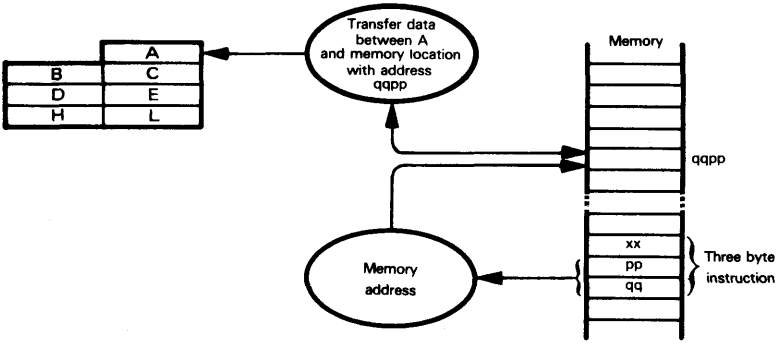
## 8080A AND 8085 MEMORY ADDRESSING

**When addressing data memory you may use implied memory addressing or direct memory addressing.** In **implied memory addressing,** the H and L registers hold the address of (or identify) the memory location being accessed. Data may be moved between the identified memory location and any one of the seven CPU registers A, B, C, D, E, H or L. This **may be illustrated as follows:**



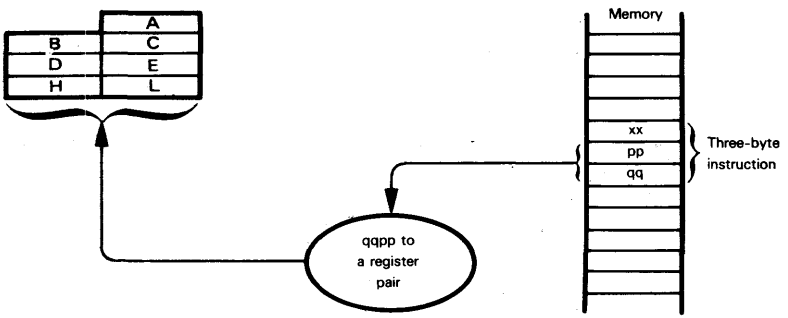

**A limited number of instructions use Registers B and C, or D and E as the Data Pointer.** These instructions move data between the Accumulator and the memory location addressed by Registers B and C or Registers D and E.

**There are also a limited number of instructions which use direct memory addressing to transfer data between the Accumulator and memory.** These are three byte instructions which may be illustrated as follows:




**Since the Stack is part of the Read/Write memory, we must consider Push and Pop instructions as Memory Reference instructions.** These instructions move two bytes of data between a register pair and the address in the Stack Pointer, i.e., current top-of-stack. The 8080A/8085 stack address is decremented with each Push and incremented with each Pop.

Some texts identify Immediate instructions as Memory Reference instructions. An Immediate instruction is a two- or three-byte instruction in which the second and third byte hold fixed data which is loaded into the Accumulator or some other CPU register. This may be illustrated as follows:




All 8080A and 8085 Jump instructions use absolute, direct addressing; that is to say, all Jump instructions are three bytes long, with the second and third bytes containing the address of the memory location from which the processor will fetch the next instruction to be executed.

Table 3-1
Frequently Used Instructions Of The 8080A And 8085

| Instruction Code | Meaning |
|---|---|
| ADC,ACI | ADD WITH CARRY |
| ADD, ADI | ADD |
| ANA, ANI | LOGICAL AND |
| CALL | CALL SUBROUTINE |
| CMP, CPI | COMPARE |
| DCR | DECREMENT |
| IN | INPUT |
| INR | INCREMENT |
| INX | INCREMENT 16 BITS |
| JC | JUMP ON CARRY |
| JMP | JUMP |
| JNC | JUMP ON NOT CARRY |
| JNZ | JUMP ON NOT ZERO |
| JZ | JUMP ON ZERO |
| LDA | LOAD ACCUMULATOR |
| LXI | LOAD 16 BITS |
| MOV | MOVE |
| MVI | MOVE IMMEDIATE |
| OUT | OUTPUT |
| RAL | ROTATE WITH CARRY LEFT |
| RAR | ROTATE WITH CARRY RIGHT |
| RET | RETURN FROM SUBROUTINE |
| STA | STORE ACCUMULATOR |
| SUB, SUI | SUBTRACT |

Table 3-2
Occasionally Used Instructions Of The 8080A And 8085

| Instruction Code | Meaning |
|---|---|
| CMA | COMPLEMENT ACCUMULATOR |
| DAA | DECIMAL ADJUST |
| DAD | 16-BIT ADD |
| DCX | 16-BIT DECREMENT |
| DI | DISABLE INTERRUPTS |
| EI | ENABLE INTERRUPTS |
| HLT | HALT |
| JM | JUMP ON MINUS |
| JP | JUMP ON POSITIVE |
| LDAX | LOAD ACCUMULATOR INDIRECT |
| LHLD | LOAD H AND L DIRECT |
| NOP | NO OPERATION |
| ORA, ORI | LOGICAL OR |
| POP | REMOVE FROM STACK |
| PUSH | ENTER INTO STACK |
| RIM (8085 ONLY) | RESET INTERRUPT MASK |
| RLC | ROTATE LEFT |
| RRC | ROTATE RIGHT |
| SHLD | STORE H AND L DIRECT |
| SIM (8085 ONLY) | SET INTERRUPT MASK |
| STAX | STORE ACCUMULATOR INDIRECT |
| XCHG | EXCHANGE D AND E, H AND L |
| XRA, XRI | LOGICAL EXCLUSIVE OR |

Table 3-3
Seldom Used Instructions Of The 8080A And 8085

| Instruction Code | Meaning |
|---|---|
| CC | CALL ON CARRY |
| CM | CALL ON MINUS |
| CMC | COMPLEMENT CARRY |
| CNC | CALL ON NO CARRY |
| CNZ | CALL ON NOT ZERO |
| CP | CALL ON POSITIVE |
| CPE | CALL ON PARITY EVEN |
| CPO | CALL ON PARITY ODD |
| CZ | CALL ON ZERO |
| JPE | JUMP ON PARITY EVEN |
| JPO | JUMP ON PARITY ODD |
| PCHL | H AND L TO PROGRAM COUNTER |
| RC | RETURN ON CARRY |
| RM | RETURN ON MINUS |
| RNC | RETURN ON NO CARRY |
| RNZ | RETURN ON NOT ZERO |
| RP | RETURN ON POSITIVE |
| RPE | RETURN ON PARITY EVEN |
| RPO | RETURN ON PARITY ODD |
| RST | RESTART |
| RZ | RETURN ON ZERO |
| SBB, SBI | SUBTRACT WITH BORROW |
| SPHL | H AND L TO STACK POINTER |
| STC | SET CARRY |
| XTHL | EXCHANGE TOP OF STACK, H AND L |

**Instructions falsely frighten microcomputer users who are new to programming. Taken as an isolated event, operations associated with the execution of a single instruction are easy enough to follow — and that is the purpose of this chapter.**

Why are the instructions of a microcomputer referred to as an instruction "set"? Because the instructions selected by the designers of any microcomputer are selected with great care; it must be easy to execute complex operations as a sequence of simple events — each of which is represented by one instruction from a well-designed instruction "set".

**Remaining consistent with An Introduction To Microcomputers: Volume II, Table 3-4 summarizes the 8080A/8085 microcomputer instruction set, with similar instructions grouped together.**

**Individual instructions are described next in alphabetical order of instruction mnemonic.**

In addition to simply stating what each instruction does, the purpose of the instruction within normal programming logic is identified.

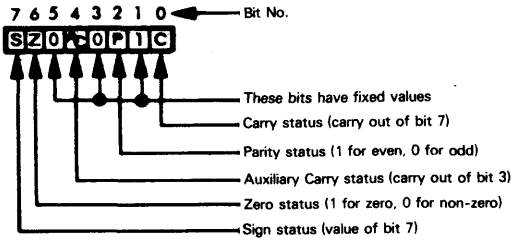
## ABBREVIATIONS

**These are the abbreviations used in this chapter:**

| | |
|---|---|
| A | The Accumulator |
| B | The B register } These are sometimes referred to as a register pair |
| C | The C register } |
| D | The D register } These are sometimes referred to as a register pair |
| E | The E register } |
| H | The H register } This register pair provides the implied memory |
| L | The L register } address |
| CS | Carry status |
| $A_C$ | Auxiliary carry status |
| ZS | Zero status |
| SS | Sign status |
| PS | Parity status |
| I | The Instruction register |
| I2 | Second object code byte |
| I3 | Third object code byte |
| PC | The Program Counter |
| SP | The Stack Pointer |
| PSW | The Program Status Word, which has bits assigned to status flags as shown on the next page |
| H | Appearing at the end of a group of digits (e.g., 213AH) specifies hexadecimal digits |
| data | 8-bit immediate data |
| dev | An I/O device |
| data16 | 16-bit immediate data |
| reg | Register A, B, C, D, E, H or L |
| M | Memory, address implied by HL |
| label | A 16-bit address, specifying an instruction label |
| rp | A register pair: B for BC, D for DE, H for HL, SP for Stack Pointer, PSW for status flags and Accumulator |
| port | An I/O port, identified by a number between 0 and $FF_{16}$ |
| addr | A 16-bit address, specifying a data memory byte |
| [ ] | Contents of location identified within brackets |
| [[ ]] | Memory byte addressed by location identified within brackets |
| ← | Move data in direction of arrow |
| ←→ | Exchange contents of locations on either side of arrow |
| + | Add |
| — | Subtract |
| •,Λ | AND |
| V | OR |
| ⊕,⊻ | XOR |

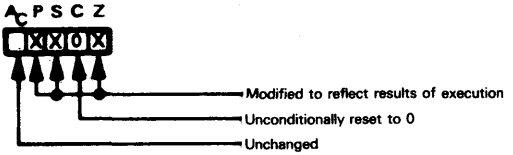
## STATUS

**The five status flags are stored in a Program Status Word (PSW) as follows:**


7 6 5 4 3 2 1 0
Bit No.
S Z 0 A 0 P 1 C
These bits have fixed values
Carry status (carry out of bit 7)
Parity status (1 for even, 0 for odd)
Auxiliary Carry status (carry out of bit 3)
Zero status (1 for zero, 0 for non-zero)
Sign status (value of bit 7)

PSW and A are sometimes treated as a register pair.

**The effect of instruction execution on status is illustrated as follows:**




Within instruction execution illustrations, an X identifies a status that is set or reset. A 0 identifies a status that is always cleared. A blank means the status does not change.

**STATUS CHANGES WITH INSTRUCTION EXECUTION**

## INSTRUCTION MNEMONICS

**The fixed part of an assembly language instruction is shown in UPPER CASE.**

**The variable part (immediate data, I/O device number, register name, label or address) is shown in lower case.**

## INSTRUCTION OBJECT CODES

**Instruction object codes are represented as two hexadecimal digits for instructions without variations.**

**Instruction object codes are represented as eight binary digits for instructions with variations; the binary digit representation of variations is then identifiable.**

## INSTRUCTION EXECUTION TIMES AND CODES

**Table 3-5 lists instructions in alphabetical order, showing object codes and execution times expressed as machine cycles.**

**Where two instruction cycles are shown, the first is for "condition not met" whereas the second is for "condition met".**

Table 3-4. A Summary Of 8080A/8085 Microcomputer Instruction Set

| Type | Mnemonic | Operand(s) | Bytes | Status | | | | | Operation Performed |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C | $A_C$ | Z | S | P | |
| I/O | IN | DEV | 2 | | | | | | [A] ← [DEV]<br>Input to A from device DEV (DEV = 0 to 255). |
| | OUT | DEV | 2 | | | | | | [DEV] ←[A]<br>Output from A to device DEV (DEV = 0 to 255). |
| Primary Memory Reference | LDAX | RP | 1 | | | | | | [A] ← [[RP]]<br>Load A using address implied by BC (RP = B) or DE (RP = D). |
| | STAX | RP | 1 | | | | | | [[RP]] ← [A]<br>Store A using implied addressing as for LDAX. |
| | MOV | R,M | 1 | | | | | | [R] ← [[HL]]<br>Load any register using address implied by HL. |
| | MOV | M,R | 1 | | | | | | [[HL]] ← [R]<br>Store any register using address implied by HL. |
| | LDA | ADDR | 3 | | | | | | [A] ← [ADDR], i.e., [A] ← [[I3,I2]]<br>Load A, use direct addressing. |
| | STA | ADDR | 3 | | | | | | [ADDR] ← [A], i.e., [[I3,I2]] ← [A]<br>Store A, use direct addressing. |
| | LHLD | ADDR | 3 | | | | | | [L] ← [ADDR], [H] ← [ADDR + 1] i.e., [L] ← [[I3,I2]], [H] ← [[I3,I2] + 1]<br>Load H and L registers, use direct addressing. |
| | SHLD | ADDR | 3 | | | | | | [ADDR] ← [L], [ADDR + 1] ← [H] i.e., [[I3,I2]] ← [L], [[I3,I2] + 1] ← [H]<br>Store H and L registers, use direct addressing. |

Table 3-4. A Summary Of 8080A/8085 Microcomputer Instruction Set (Continued)

| Type | Mnemonic | Operand(s) | Bytes | Status | | | | | Operation Performed |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C | $AC_*$ | Z | S | P | |
| Secondary Memory Reference (Memory Operate) | ADD | M | 1 | X | X | X | X | X | [A] ← [A] + [[HL]]<br>Add to A |
| | ADC | M | 1 | X | X | X | X | X | [A] ← [A] + [[HL]] + [CS]<br>Add with Carry to A |
| | SUB | M | 1 | X | X | X | X | X | [A] ← [A] - [[HL]]<br>Subtract from A |
| | SBB | M | 1 | X | X | X | X | X | [A] ← [A] - [[HL]] - [CS]<br>Subtract from A with borrow |
| | ANA | M | 1 | 0 | X** | X | X | X | [A] ← [A] ∧ [[HL]]<br>AND with A |
| | XRA | M | 1 | 0 | 0 | X | X | X | [A] ← [A] ⊻ [[HL]]<br>Exclusive-OR with A |
| | ORA | M | 1 | 0 | 0 | X | X | X | [A] ← [A] ∨ [[HL]]<br>OR with A |
| | CMP | M | 1 | X | X | X | X | X | [A] - [[HL]]<br>Compare with A |
| | INR | M | 1 | | X | X | X | X | [[HL]] ← [[HL]] + 1<br>Increment memory |
| | DCR | M | 1 | | X | X | X | X | [[HL]] ← [[HL]] - 1<br>Decrement memory |
| Immediate | LXI | RP,DATA16 | 3 | | | | | | [RP] ← DATA16<br>Load 16-bit immediate data into BC (RP = B), DE (RP = D), HL (RP = H) or SP (RP = SP) |
| | MVI | M,DATA | 2 | | | | | | [[HL]] ← DATA<br>Load 8-bit immediate data into memory location with address implied by HL. |
| | MVI | R,DATA | 2 | | | | | | [R] ← DATA<br>Load 8-bit immediate data into any register |

Table 3-4. A Summary Of 8080A/8085 Microcomputer Instruction Set (Continued)

| Type | Mnemonic | Operand(s) | Bytes | Status | | | | | Operation Performed |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C | $A_C$ | Z | S | P | |
| Jump | JMP | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump to instruction with label ADDR. |
| | PCHL | | 1 | | | | | | [PC] ← [HL]<br>Jump to instruction at address contained in HL. |
| Subroutine Call and Return (Immediate and Stack) | CALL | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine starting at ADDR |
| | CC | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine if C = 1 |
| | CNC | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine if C = 0 |
| | CZ | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine if Z = 1 |
| | CNZ | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine if Z = 0 |
| | CP | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine if S = 0 |
| | CM | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine if S = 1 |
| | CPE | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine if even parity |
| | CPO | ADDR | 3 | | | | | | [[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2<br>Jump to subroutine if odd parity. |
| | RET | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine |

Table 3-4. A Summary Of 8080A/8085 Microcomputer Instruction Set (Continued)

| Type | Mnemonic | Operand(s) | Bytes | Status | | | | | Operation Performed |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C | $A_C$ | Z | S | P | |
| Subroutine Call and Return (Immediate and Stack) | RC | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine if C = 1 |
| | RNC | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine if C = 0 |
| | RZ | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine if Z = 1 |
| | RNZ | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine if Z = 0 |
| | RM | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine if S = 1 |
| | RP | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine if S = 0 |
| | RPE | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine if even parity |
| | RPO | | 1 | | | | | | [PC] ← [[SP]], [SP] ← [SP] + 2<br>Return from subroutine if odd parity |
| Immediate Operate | ADI | DATA | 2 | X | X | X | X | X | [A] ← [A] + DATA<br>Add immediate to A |
| | ACI | DATA | 2 | X | X | X | X | X | [A] ← [A] + DATA + [CS]<br>Add with carry immediate to A |
| | SUI | DATA | 2 | X | X | X | X | X | [A] ← [A] - DATA<br>Subtract immediate from A |
| | SBI | DATA | 2 | X | X | X | X | X | [A] ← [A] - DATA - [CS]<br>Subtract immediate with borrow from A |
| | ANI | DATA | 2 | 0 | X** | X | X | X | [A] ← [A] ∧ DATA<br>AND immediate with A |

Table 3-4. A Summary Of 8080A/8085 Microcomputer Instruction Set (Continued)

| Type | Mnemonic | Operand(s) | Bytes | Status | | | | | Operation Performed |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C | $A_C$ | Z | S | P | |
| Immediate Operate | XRI | DATA | 2 | 0 | 0 | X | X | X | [A] ← [A] ⊻ DATA<br>Exclusive-OR immediate with A |
| | ORI | DATA | 2 | 0 | 0 | X | X | X | [A] ← [A] V DATA<br>OR immediate with A |
| | CPI | DATA | 2 | X | X | X | X | X | [A] - DATA<br>Compare immediate with A |
| Jump On Condition | JC | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump if C = 1 |
| | JNC | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump if C = 0 |
| | JZ | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump if Z = 1 |
| | JNZ | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump if Z = 0 |
| | JP | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump if S = 0 |
| | JM | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump if S = 1 |
| | JPE | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump on even parity |
| | JPO | ADDR | 3 | | | | | | [PC] ← ADDR<br>Jump on odd parity |

Table 3-4. A Summary Of 8080A/8085 Microcomputer Instruction Set (Continued)

| Type | Mnemonic | Operand(s) | Bytes | Status | | | | | Operation Performed |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C | AC | Z | S | P | |
| Reg-Reg Move | MOV | D,S | 1 | | | | | | [R] ← [R]<br>Move any register (S) to any register (D) |
| | XCHG | | 1 | | | | | | [D] ⟷ [H], [E] ⟷ [L]<br>Exchange DE with HL |
| | SPHL | | 1 | | | | | | [HL] → [SP]<br>Move HL to SP |
| Register-Register Operate | ADD | R | 1 | X | X | X | X | X | [A] ← [A] + [R]<br>Add any register to A |
| | ADC | R | 1 | X | X | X | X | X | [A] ← [A] + [R] + [CS]<br>Add with Carry any register to A |
| | SUB | R | 1 | X | X | X | X | X | [A] ← [A] - [R]<br>Subtract any register from A |
| | SBB | R | 1 | X | X | X | X | X | [A] ← [A] - [R] - [CS]<br>Subtract any register with borrow from A |
| | ANA | R | 1 | 0 | X** | X | X | X | [A] ← [A] ∧ [R]<br>AND any register with A |
| | XRA | R | 1 | 0 | 0 | X | X | X | [A] ← [A] ⊻ [R]<br>Exclusive-OR any register with A |
| | ORA | R | 1 | 0 | 0 | X | X | X | [A] ← [A] ∨ [R]<br>OR any register with A |
| | CMP | R | 1 | X | X | X | X | X | [A] - [R] Compare any register with A |
| Register Operate | INR | R | 1 | | X | X | X | X | [R] ← [R] + 1<br>Increment any register |
| | DCR | R | 1 | | X | X | X | X | [R] ← [R] - 1<br>Decrement any register |
| | CMA | | 1 | | | | | | [A] ← [$\overline{A}$] Complement A |

Table 3-4. A Summary Of 8080A/8085 Microcomputer Instruction Set (Continued)

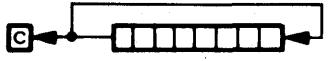
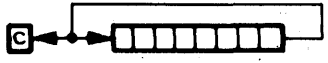
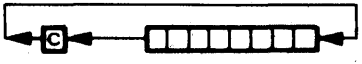
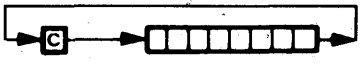
| Type | Mnemonic | Operand(s) | Bytes | Status | | | | | Operation Performed |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C | $A_C$ | Z | S | P | |
| Register Operate | DAA | | 1 | X | X | X | X | X | Decimal adjust A |
| | RLC | | 1 | X | | | | |  Rotate A left with branch carry |
| | RRC | | 1 | X | | | | |  Rotate A right with branch carry |
| | RAL | | 1 | X | | | | |  Rotate A left with carry |
| | RAR | | 1 | X | | | | |  Rotate A right with carry |
| | DAD | RP | 1 | X | | | | | [HL] ← [HL] + [RP]<br>Add RP to HL. RP=BC, DE, HL or SP. |
| | INX | RP | 1 | | | | | | [RP] ← [RP] + 1<br>Increment RP. RP = BC, DE, HL or SP. |
| | DCX | RP | 1 | | | | | | [RP] ← [RP] - 1<br>Decrement RP. RP=BC, DE, HL or SP. |
| Stack | PUSH | RP | 1 | | | | | | [[SP]] ← [RP], [SP] ← [SP] - 2<br>Push RP contents onto stack |
| | POP | RP | 1 | | | | | | [RP] ← [[SP]], [SP] ← [SP] + 2<br>Pop stack into RP<br>RP=BC, DE, HL or SP |
| | XTHL | | 1 | | | | | | [HL] ⟷ [[SP]]<br>Exchange HL with top of stack |

Table 3-4. A Summary Of 8080A/8085 Microcomputer Instruction Set (Continued)

| Type | Mnemonic | Operand(s) | Bytes | Status | | | | | Operation Performed |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C | $A_C$ | Z | S | P | |
| Interrupt | EI | | 1 | | | | | | Enable interrupts following execution of the next instruction |
| | RIM | | 1 | | | | | | Read interrupt mask* |
| | DI | | 1 | | | | | | Disable interrupts |
| | SIM | | 1 | | | | | | Set interrupt mask* |
| | RST | | 1 | | | | | | Restart |
| Status | STC | | 1 | 1 | | | | | [CS] ← 1<br>Set Carry |
| | CMC | | 1 | X | | | | | [CS] ← [CS]<br>Complement Carry |
| | NOP | | 1 | | | | | | |
| | HLT | | 1 | | | | | | |

Statuses:
- C = Carry
- $A_C$ = Carry out of bit 3
- Z = Zero
- S = Sign
- P = Parity
- X = Status set or reset
- 0 = Status reset

Blank = Status unchanged
*8085 instructions
**8085 sets $A_C$ to 1 for all AND instructions

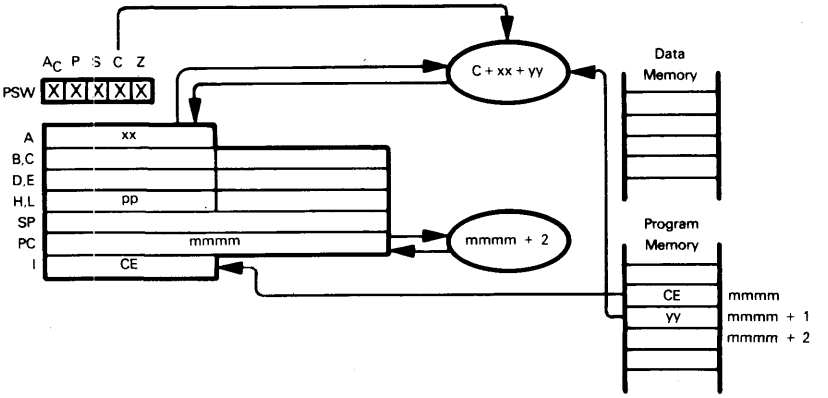
Table 3-5. A Summary Of Instruction Object Codes And Execution Cycles

| Instruction | | Object Code | Bytes | Cycles | Instruction | | Object Code | Bytes | Cycles |
|---|---|---|---|---|---|---|---|---|---|
| ACI | DATA | CE yy | 2 | 7 | LXI | RP,DATA16 | 00xx0001 | 3 | 10 |
| ADC | REG | 10001xxx | 1 | 4 | | | yyyy | | |
| ADC | M | 8E | 1 | 7 | MOV | REG,REG | 01dddsss | 1 | 5 |
| ADD | REG | 10000xxx | 1 | 4 | MOV | M,REG | 01110sss | 1 | 7 |
| ADD | M | 86 | 1 | 7 | MOV | REG,M | 01ddd110 | 1 | 7 |
| ADI | DATA | C6 yy | 2 | 7 | MVI | REG,DATA | 00ddd110 | 2 | 7 |
| ANA | REG | 10100xxx | 1 | 4 | | | yy | | |
| ANA | M | A6 | 1 | 7 | MVI | M,DATA | 36 yy | 2 | 10 |
| ANI | DATA | E6 yy | 2 | 7 | NOP | | 00 | 1 | 4 |
| CALL | LABEL | CD ppqq | 3 | 17 | ORA | REG | 10110xxx | 1 | 4 |
| CC | LABEL | DC ppqq | 3 | 11/17 | ORA | M | B6 | 1 | 7 |
| CM | LABEL | FC ppqq | 3 | 11/17 | ORI | DATA | F6 yy | 2 | 7 |
| CMA | | 2F | 1 | 4 | OUT | PORT | D3 yy | 2 | 10 |
| CMC | | 3F | 1 | 4 | PCHL | | E9 | 1 | 5 |
| CMP | REG | 10111xxx | 1 | 4 | POP | RP | 11xx0001 | 1 | 10 |
| CMP | M | BE | 1 | 7 | PUSH | RP | 11xx0101 | 1 | 11 |
| CNC | LABEL | D4 ppqq | 3 | 11/17 | RAL | | 17 | 1 | 4 |
| CNZ | LABEL | C4 ppqq | 3 | 11/17 | RAR | | 1F | 1 | 4 |
| CP | LABEL | F4 ppqq | 3 | 11/17 | RC | | D8 | 1 | 5/11 |
| CPE | LABEL | EC ppqq | 3 | 11/17 | RET | | C9 | 1 | 10 |
| CPI | DATA | FE yy | 2 | 7 | RIM | | 20 | 1 | 4* |
| CPO | LABEL | E4 ppqq | 3 | 11/17 | RLC | | 07 | 1 | 4 |
| CZ | LABEL | CC ppqq | 3 | 11/17 | RM | | F8 | 1 | 5/11 |
| DAA | | 27 | 1 | 4 | RNC | | D0 | 1 | 5/11 |
| DAD | RP | 00xx1001 | 1 | 10 | RNZ | | C0 | 1 | 5/11 |
| DCR | REG | 00xxx101 | 1 | 5 | RP | | F0 | 1 | 5/11 |
| DCR | M | 35 | 1 | 10 | RPE | | E8 | 1 | 5/11 |
| DCX | RP | 00xx1011 | 1 | 5 | RPO | | E0 | 1 | 5/11 |
| DI | | F3 | 1 | 4 | RRC | | 0F | 1 | 4 |
| EI | | FB | 1 | 4 | RST | N | 11xxx111 | 1 | 11 |
| HLT | | 76 | 1 | 7 | RZ | | C8 | 1 | 5/11 |
| IN | PORT | DB yy | 2 | 10 | SBB | REG | 10011xxx | 1 | 4 |
| INR | REG | 00xxx100 | 1 | 5 | SBB | M | 9E | 1 | 7 |
| INR | M | 34 | 1 | 10 | SBI | DATA | DE yy | 2 | 7 |
| INX | RP | 00xx0011 | 1 | 5 | SHLD | ADDR | 22 ppqq | 3 | 16 |
| JC | LABEL | DA ppqq | 3 | 10 | SIM | | 30 | 1 | 4* |
| JM | LABEL | FA ppqq | 3 | 10 | SPHL | | F9 | 1 | 5 |
| JMP | LABEL | C3 ppqq | 3 | 10 | STA | ADDR | 32 ppqq | 3 | 13 |
| JNC | LABEL | D2 ppqq | 3 | 10 | STAX | RP | 000x0010 | 1 | 7 |
| JNZ | LABEL | C2 ppqq | 3 | 10 | STC | | 37 | 1 | 4 |
| JP | LABEL | F2 ppqq | 3 | 10 | SUB | REG | 10010xxx | 1 | 4 |
| JPE | LABEL | EA ppqq | 3 | 10 | SUB | M | 96 | 1 | 7 |
| JPO | LABEL | E2 ppqq | 3 | 10 | SUI | DATA | D6 yy | 2 | 7 |
| JZ | LABEL | CA ppqq | 3 | 10 | XCHG | | EB | 1 | 4 |
| LDA | ADDR | 3A ppqq | 3 | 13 | XRA | REG | 10101xxx | 1 | 4 |
| LDAX | RP | 000x1010 | 1 | 7 | XRA | M | AE | 1 | 7 |
| LHLD | ADDR | 2A ppqq | 3 | 16 | XRI | DATA | EE yy | 2 | 7 |
| | | | | | XTHL | | E3 | 1 | 18 |

| | |
|---|---|
| ppqq | represents four hexadecimal digit memory address |
| yy | represents two hexadecimal data digits |
| yyyy | represents four hexadecimal data digits |
| x | represents an optional binary digit |
| ddd | represents optional binary digits identifying a destination register |
| sss | represents optional binary digits identifying a source register |

*8085 instructions

## ACI — ADD WITH CARRY IMMEDIATE TO ACCUMULATOR




ACI (CE) data (yy)

Add the contents of the next program memory byte and the Carry status to the Accumulator.

Suppose xx = $3A_{16}$, yy = $7C_{16}$, C = 0. After the instruction:

ACI 7CH

has executed, the Accumulator will contain B6:

```
3A  =  00111010
7C  =  01111100
Carry  =       0
       --------
       10110110
```

No carry sets C to 0

1 sets S to 1

Five 1 bits, set P to 0

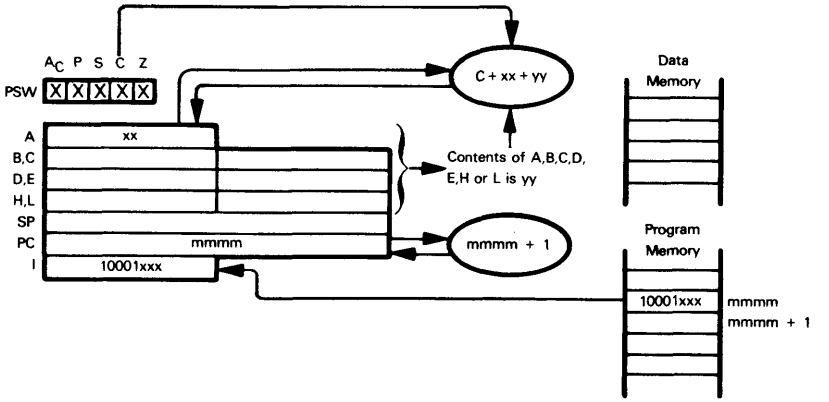
Non-zero result, set Z to 0

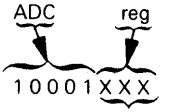
Carry sets $A_C$ to 1

This is a routine data manipulation instruction.

## ADC — ADD REGISTER OR MEMORY WITH CARRY TO ACCUMULATOR

This instruction takes two forms. First consider a register's contents added to the Accumulator:



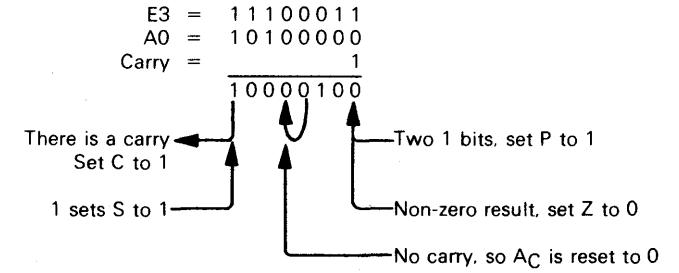

000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Add the contents of register A, B, C, D, E, H or L and the Carry status to the Accumulator.

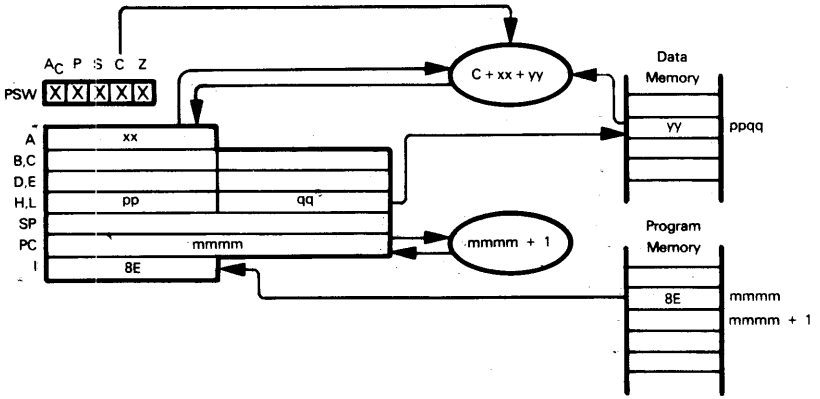
Suppose xx = $E3_{16}$, register E contains $A0_{16}$, C = 1. After instruction:

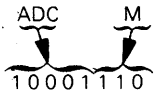
ADC E

has executed, the Accumulator will contain $84_{16}$:


E3 = 11100011
A0 = 10100000
Carry = 1
10000100
There is a carry
Set C to 1
1 sets S to 1
Two 1 bits, set P to 1
Non-zero result, set Z to 0
No carry, so $A_C$ is reset to 0

The contents of a memory byte may also be added, with Carry, to the Accumulator:




If xx = $E3_{16}$, yy = $A0_{16}$ and C = 1, then execution of the instruction:

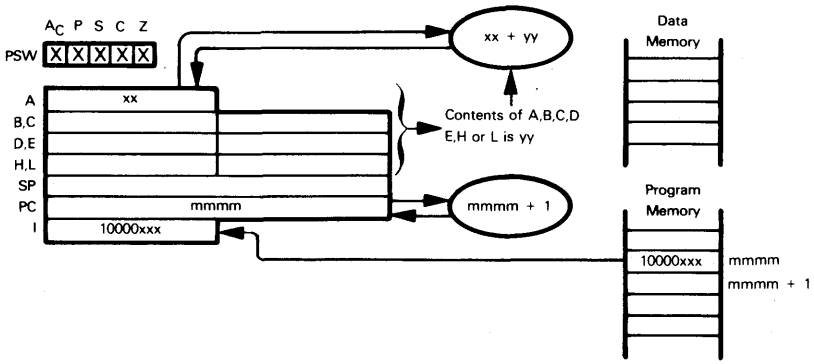
ADC M

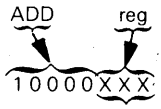
generates the same result as execution of the ADC E instruction, which was just described.

The ADC instruction is most frequently used in multibyte addition, for the second and subsequent bytes.

## ADD — ADD REGISTER OR MEMORY TO ACCUMULATOR

This instruction takes two forms. First consider a register's contents added to the Accumulator:




0 0 0 for reg = B
0 0 1 for reg = C
0 1 0 for reg = D
0 1 1 for reg = E
1 0 0 for reg = H
1 0 1 for reg = L
1 1 1 for reg = A

Add the contents of register A, B, C, D, E, H or L to the Accumulator.

Suppose xx = $E3_{16}$, register E contains $A0_{16}$, C = 1. After the instruction:

```
ADD     E
```

has executed, the Accumulator will contain $83_{16}$:

```
E3 =  11100011
A0 =  10100000
      10000011
```

There is a carry — Set C to 1

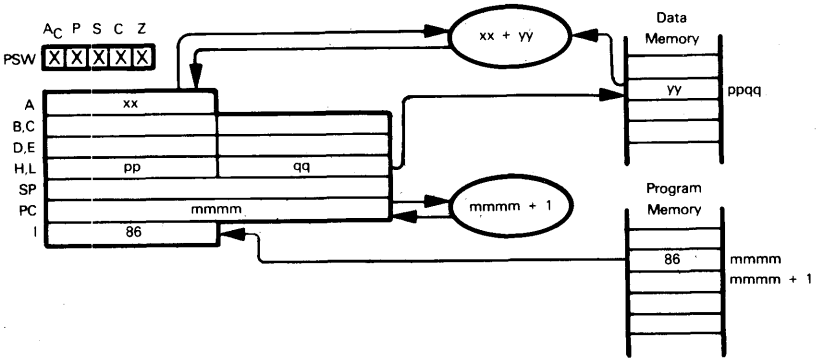
1 sets S to 1

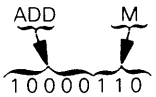
Three 1 bits, set P to 0

Non-zero result, set Z to 0

No carry, so $A_C$ is reset to 0

The contents of a memory byte may also be added to the Accumulator:



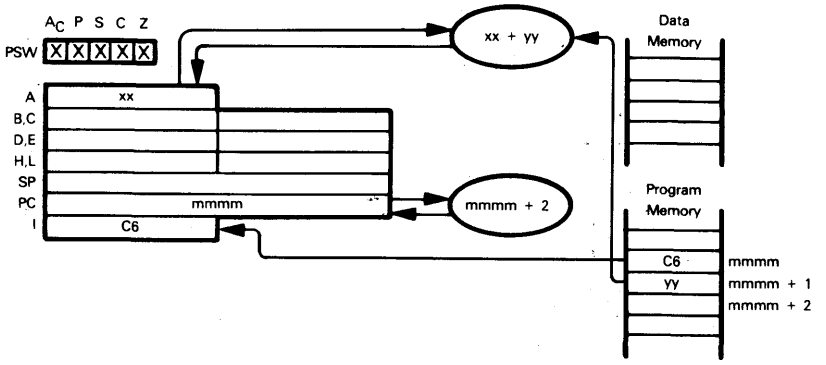

If xx = $E3_{16}$, yy = $A0_{16}$, and C = 1, then execution of the instruction:

ADD M

generates the same result as execution of the ADD E instruction, which was just described.

ADD is the binary addition instruction used in normal, single-byte operations; it is also the instruction used to add the low-order bytes of two multibyte numbers.

## ADI — ADD IMMEDIATE TO ACCUMULATOR



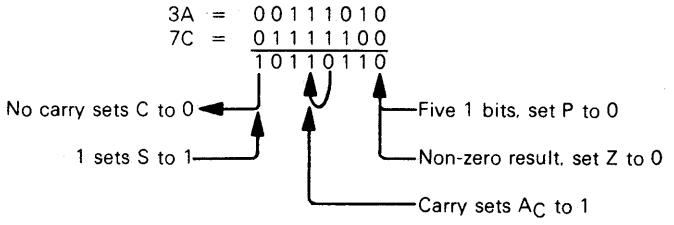

ADI C6 data yy

Add the contents of the next program memory byte to the Accumulator.

Suppose xx = $3A_{16}$, yy = $7C_{16}$, C = 0. After the instruction:

ADI 7CH

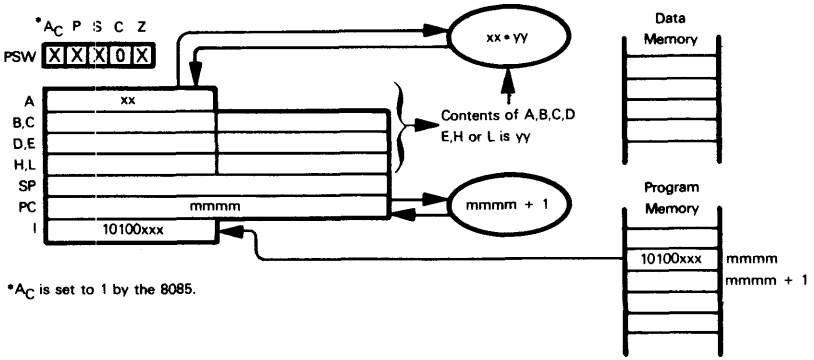
has executed, the Accumulator will contain B6:


3A = 0 0 1 1 1 0 1 0
7C = 0 1 1 1 1 1 0 0
1 0 1 1 0 1 1 0
No carry sets C to 0
1 sets S to 1
Five 1 bits, set P to 0
Non-zero result, set Z to 0
Carry sets AC to 1

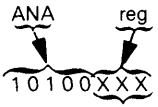
This is a routine data manipulation instruction.

## ANA — AND REGISTER OR MEMORY WITH ACCUMULATOR

This instruction takes two forms. First consider a register's contents ANDed with the Accumulator:




*$A_C$ is set to 1 by the 8085.




000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

AND the Accumulator with the contents of register A, B, C, D, E, H or L. Save the result in the Accumulator.

Suppose xx = $E3_{16}$, register E contains $A0_{16}$. After the instruction:

```
ANA    E
```

has executed, the Accumulator will contain $A0_{16}$:

```
E3 =  11100011
A0 =  10100000
      --------
      10100000
```

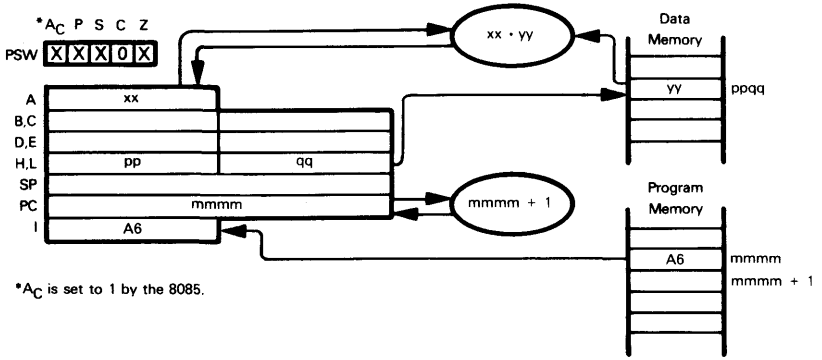
Carry is always set to 0

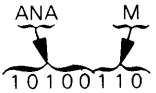
1 sets S to 1

Two 1 bits, set P to 1

Non-zero result, set Z to 0

The contents of a memory byte may also be ANDed to the Accumulator:



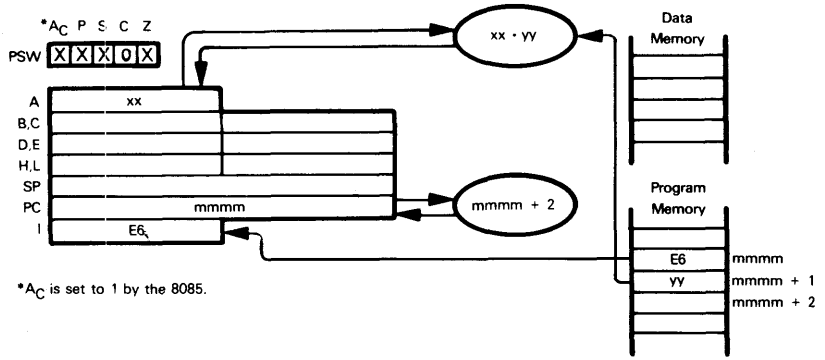




If xx = $E3_{16}$, yy = $A0_{16}$ and C = 1, then execution of the instruction:

ANA M

generates the same result as execution of the ANA E instruction, which was just described. ANA is a frequently used logical instruction.

## ANI — AND IMMEDIATE WITH ACCUMULATOR



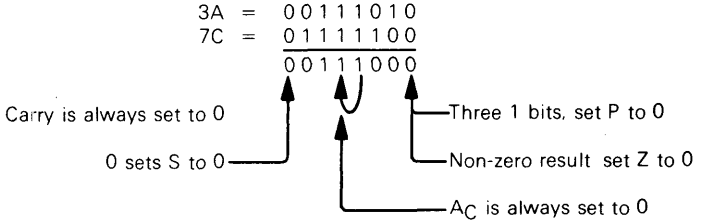

ANI E6 data yy

AND the contents of the next program memory byte to the Accumulator.

Suppose xx = $3A_{16}$, yy = $7C_{16}$. After the instruction:

ANI 7CH

has executed, the Accumulator will contain $38_{16}$.

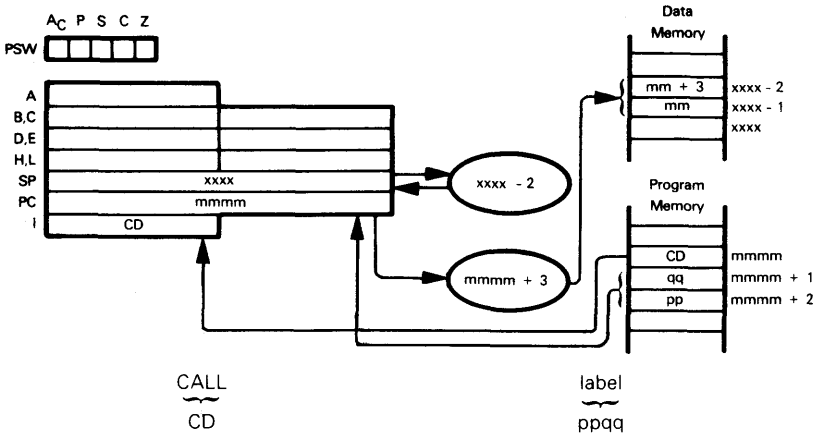

3A = 0 0 1 1 1 0 1 0
7C = 0 1 1 1 1 1 0 0
0 0 1 1 1 0 0 0
Carry is always set to 0
0 sets S to 0
Three 1 bits, set P to 0
Non-zero result set Z to 0
AC is always set to 0

This is a routine logical instruction; it is often used to turn bits "off". For example, the instruction:

ANI 7FH

will unconditionally set the high-order Accumulator bit to 0.

## CALL — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND




Store the address of the instruction following the CALL on the top of the stack; the top of the stack is a data memory byte addressed by the Stack Pointer. Then subtract 2 from the Stack Pointer in order to address the new top of stack. Move the 16-bit address contained in the second and third CALL instruction object program bytes to the Program Counter.

Consider the instruction sequence:

```
        CALL    SUBR
        ANI     7CH
        -
        -
        -
SUBR
```

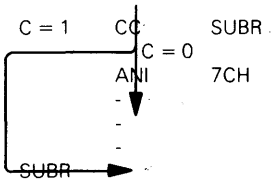
After the CALL instruction has executed, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## CC — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE CARRY STATUS EQUALS 1

| CC | label |
|---|---|
| DC | ppqq |

This instruction is identical to the CALL instruction except that the identified subroutine will be called only if the Carry status equals 1; otherwise, the instruction sequentially following the CC instruction will be executed.

Consider the instruction sequence:



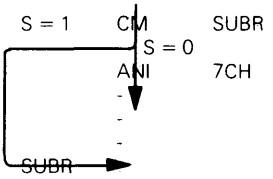

After the CC instruction has executed, if the Carry status does not equal 1 the ANI instruction will be executed. If the Carry status does equal 1, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## CM — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE SIGN STATUS EQUALS 1

| CM | label |
|---|---|
| FC | ppqq |

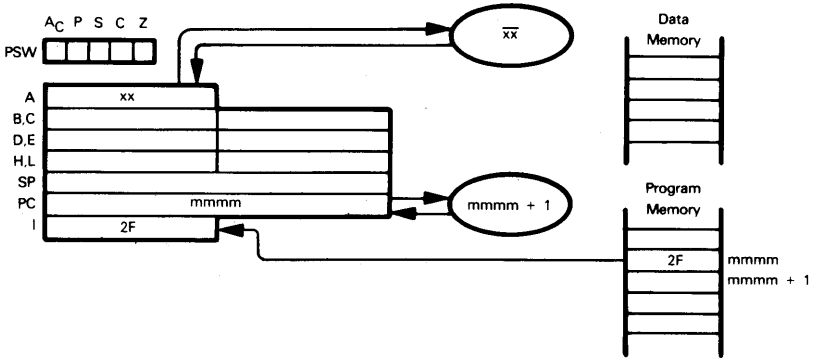
This instruction is identical to the CALL instruction except that the identified subroutine will be called only if the Sign status equals 1; otherwise, the instruction sequentially following the CM instruction will be executed.

Consider the instruction sequence:




After the CM instruction has executed, if the Sign status does not equal 1 the ANI instruction will be executed. If the Sign status does equal 1, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## CMA — COMPLEMENT THE ACCUMULATOR




CMA
2F

Complement the contents of the Accumulator. No other register contents or statuses are affected.

Suppose the Accumulator contains $3A_{16}$. After the instruction:

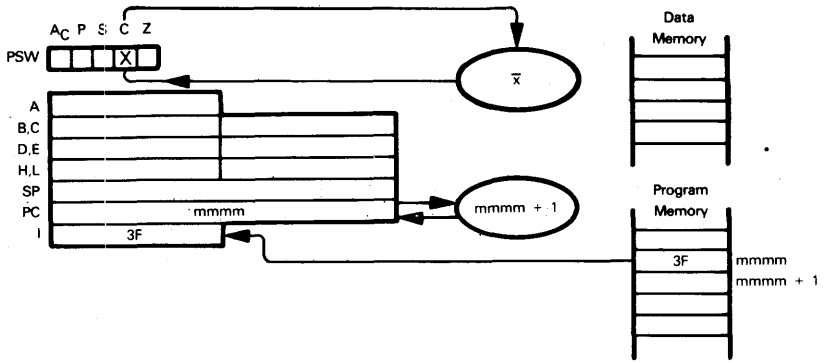
CMA

has executed, the Accumulator will contain $C5_{16}$:

$3A_{16}$ = 00111010
Complement = 11000101

This is a routine logical instruction. **Do not use it for binary subtraction.** There are special subtract instructions (SUB and SBB).

## CMC — COMPLEMENT THE CARRY STATUS




CMC
3F

Complement the Carry status. No other statuses or register contents are affected.

Suppose the Carry status contains 1. After the instruction:

CMC

has executed, the Carry status will contain 0.

This instruction is used to force the Carry status to 0, via the instruction sequence:

```
STC     ;SET CARRY STATUS TO 1
CMC     ;COMPLEMENT CARRY STATUS
```

Note you can set the Carry status to 0 via the instruction:

ANA A

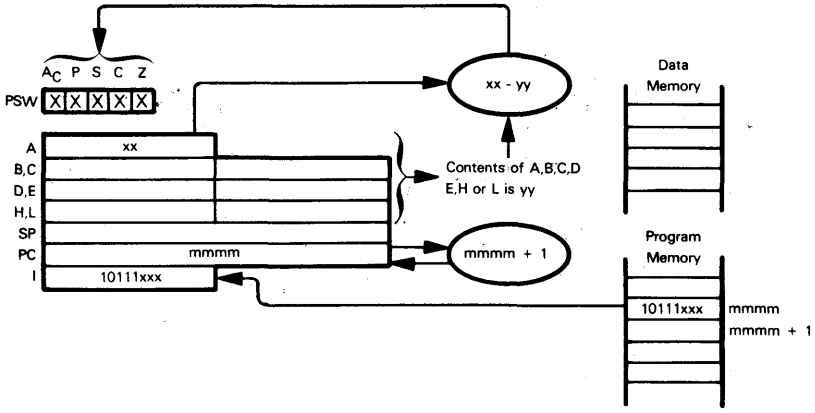
which automatically zeros the Carry status, but does not modify any register's contents since the Accumulator is ANDed with itself. The:

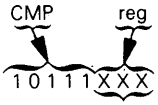
ORA A

instruction serves the same purpose.

## CMP — COMPARE REGISTER OR MEMORY WITH ACCUMULATOR

This instruction takes two forms. First consider a register's contents compared with the Accumulator:




000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Subtract the contents of register A, B, C, D, E, H or L from the contents of the Accumulator, treating both numbers as simple binary data. Discard the result, i.e., leave the Accumulator alone, but modify status flags to reflect the result of the subtraction.

Suppose xx = $E3_{16}$, register E contains $A0_{16}$. After the instruction:

```
CMP     E
```

has executed, the Accumulator will still contain $E3_{16}$, but statuses will be modified as follows:

```
E3          =   11100011
Twos comp of A0 =  01100000
                  ---------
                  01000011
```

1 ← Carry sets C to 0

0 sets S to 0

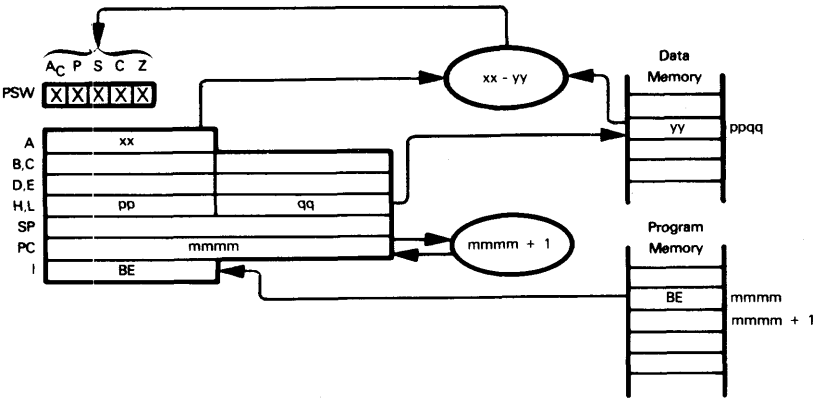
Three 1 bits, set P to 0

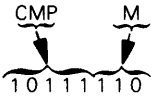
Non-zero result, set Z to 0

No carry, so $A_C$ is reset to 0

Notice that the resulting Carry is complemented.

The contents of a memory byte may also be compared with the Accumulator:




If xx = $E3_{16}$ and yy = $A0_{16}$, then execution of the instruction:

CMP M

generates the same result as execution of the CMP E instruction, which was just described.

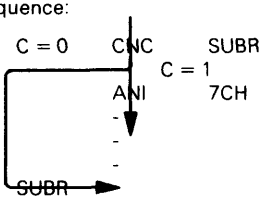
Compare instructions frequently precede conditional Call, Return and Jump instructions. The Compare Immediate (CPI) instruction is more useful than the CMP instruction.

## CNC — CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND, BUT ONLY IF THE CARRY STATUS EQUALS 0

CNC label
D4 ppqq

This instruction is identical to the CALL instruction except that the identified subroutine will be called only if the Carry status equals 0; otherwise, the instruction sequentially following the CNC instruction will be executed.

Consider the instruction sequence:



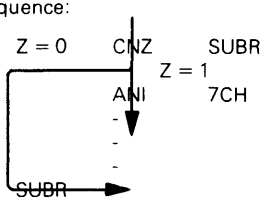

After the CNC instruction has executed, if the Carry status does not equal 0 the ANI instruction will be executed. If the Carry status does equal 0, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## CNZ — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE ZERO STATUS EQUALS 0

| CNZ | label |
|---|---|
| C4 | ppqq |

This instruction is identical to the CALL instruction except that the identified subroutine will be called only if the Zero status equals 0; otherwise, the instruction sequentially following the CNZ instruction will be executed.

Consider the instruction sequence:



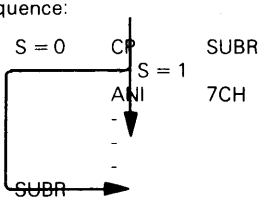

After the CNZ instruction has executed, if the Zero status does not equal 0 the ANI instruction will be executed. If the Zero status does equal 0, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## CP — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE SIGN STATUS EQUALS 0

| CP | label |
|---|---|
| F4 | ppqq |

This instruction is identical to the CALL instruction except that the identified subroutine will be called only if the Sign status equals 0; otherwise, the instruction sequentially following the CP instruction will be executed.

Consider the instruction sequence:



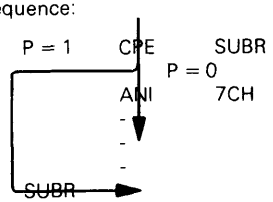

After the CP instruction has executed, if the Sign status does not equal 0 the ANI instruction will be executed. If the Sign status does equal 0, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## CPE — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE PARITY STATUS EQUALS 1

| CPE | label |
|---|---|
| EC | ppqq |

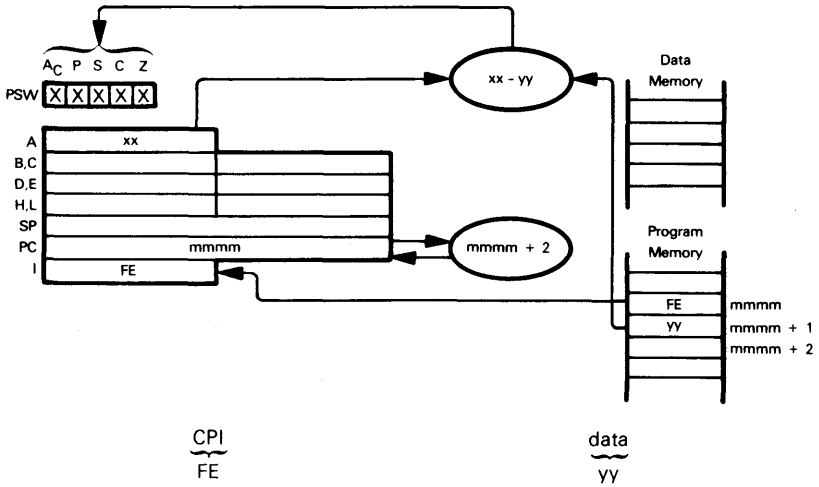
This instruction is identical to the CALL instruction except that the identified subroutine will be called only if the Parity status equals 1; otherwise, the instruction sequentially following the CPE instruction will be executed.

Consider the instruction sequence:




After the CPE instruction has executed, if the Parity status does not equal 1 the ANI instruction will be executed. If the Parity status does equal 1, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## CPI — COMPARE ACCUMULATOR CONTENTS WITH IMMEDIATE DATA

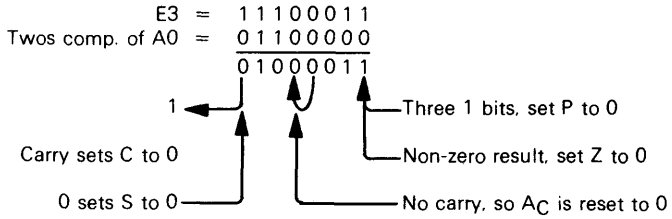



Subtract the contents of the second object code byte from the contents of the Accumulator, treating both numbers as simple, binary data. Discard the result, i.e., leave the Accumulator alone, but modify the status flags to reflect the result of the subtraction.

Suppose xx = $E3_{16}$ and the second byte of the CPI instruction object code contains $A0_{16}$. After the instruction:

CPI A0H

has executed, the Accumulator will still contain $E3_{16}$ but statuses will be modified as follows:


E3 = 1 1 1 0 0 0 1 1
Twos comp. of A0 = 0 1 1 0 0 0 0 0
0 1 0 0 0 0 1 1
1
Carry sets C to 0
0 sets S to 0
Three 1 bits, set P to 0
Non-zero result, set Z to 0
No carry, so AC is reset to 0

Notice that the resulting Carry is complemented.

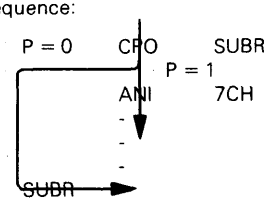
This is the instruction most frequently used to set statuses prior to execution of a conditional Call, Return or Jump instruction.

## CPO — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE PARITY STATUS EQUALS 0

| CPO | label |
|---|---|
| E4 | ppqq |

This instruction is identical to the CALL instruction except that the identified subroutine will be called only if the Parity status equals 0; otherwise, the instruction sequentially following the CPO instruction will be executed.

Consider the instruction sequence:



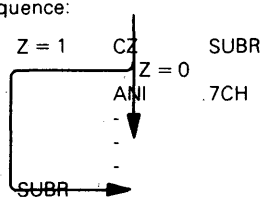

After the CPO instruction has executed, if the Parity status does not equal 0 the ANI instruction will be executed. If the Parity status does equal 0, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## CZ — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE ZERO STATUS EQUALS 1

| CZ | label |
|---|---|
| CC | ppqq |

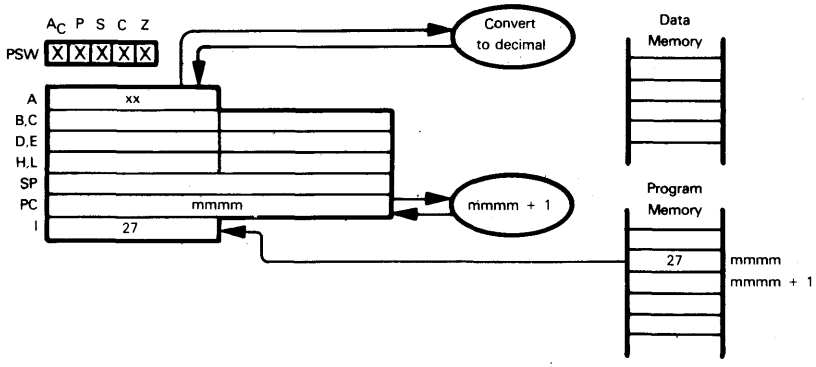
This instruction is identical to the CALL instruction except that the identified subroutine will be called only if the Zero status equals 1; otherwise, the instruction sequentially following the CZ instruction will be executed.

Consider the instruction sequence:




After the CZ instruction has executed, if the Zero status does not equal 1 the ANI instruction will be executed. If the Zero status does equal 1, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

## DAA — DECIMAL ADJUST ACCUMULATOR




DAA
27

Convert the contents of the Accumulator to its binary coded decimal form. This instruction should be used only after adding two BCD numbers, i.e., look upon ADD DAA or ADC DAA or SUB DAA or SBB DAA as compound, decimal arithmetic instructions which operate on BCD source to generate BCD answers.

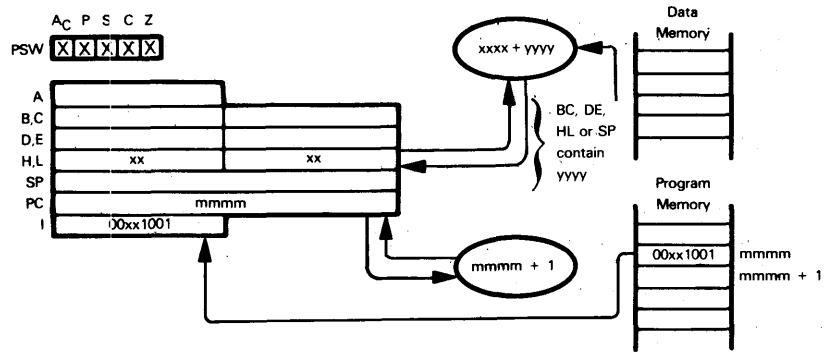
Suppose the Accumulator contains $39_{16}$ and the B register contains $47_{16}$. After the instructions:

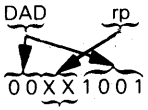
```
ADD  B
DAA
```

have executed, the Accumulator will contain $86_{16}$, not $80_{16}$.

The DAA instruction modifies all status flags, but only the Carry status is significant. Consider other statuses to have been destroyed.

## DAD — ADD A REGISTER PAIR TO H AND L




0 0 for rp = B, representing B,C
0 1 for rp = D, representing D,E
1 0 for rp = H, representing H,L
1 1 for rp = SP, representing the Stack Pointer

Add the 16-bit value from the BC, DE or HL register pair or the Stack Pointer to the HL register pair.

Suppose H,L contains $034A_{16}$ and B,C contains $214C_{16}$. After the instruction:

DAD B

has executed, the HL register pair will contain $2496_{16}$:

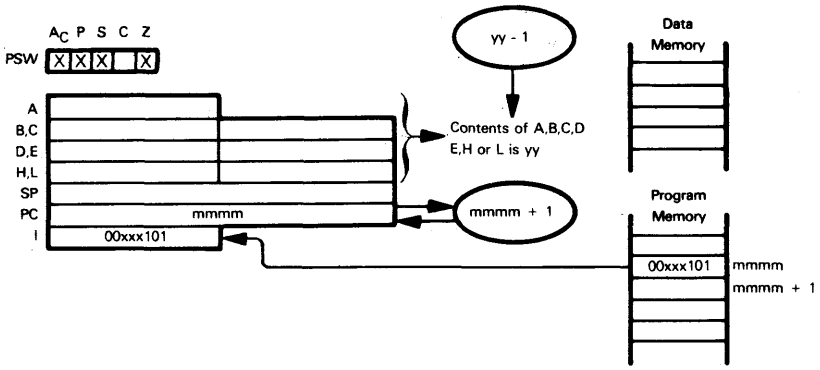
```
034A = 0000001101001010
214C = 0010000101001100
       0010010010010110
```

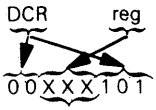
There is no carry so C is reset to 0

No other statuses are affected

The DAD instruction is one of the most useful in the 8080 instruction set for traditional programming applications. This instruction provides the equivalent of indexed addressing, and the DAD H instruction is equivalent to a 16-bit left shift.

## DCR — DECREMENT REGISTER OR MEMORY CONTENTS



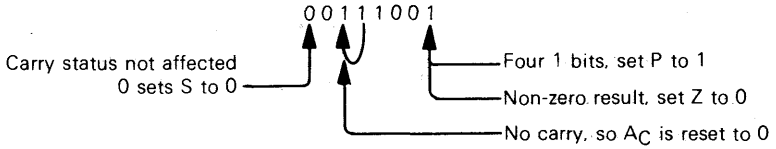

000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Subtract 1 from the contents of the specified register.

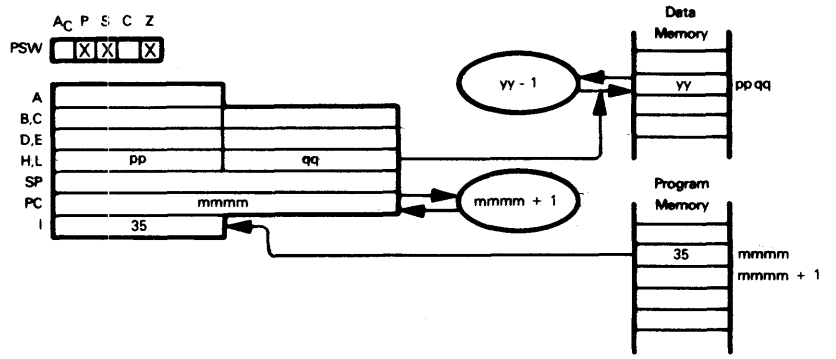
Suppose the C register contains $3A_{16}$. After the instruction:

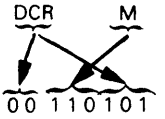
DCR C

has executed, the C register will contain $39_{16}$:



The contents of a read/write memory byte may also be decremented:




Suppose HL contains $3714_{16}$. Then execution of the instruction:

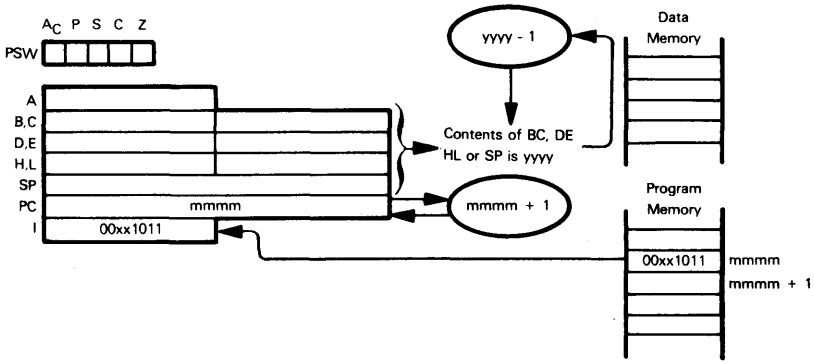
```
DCR     M
```

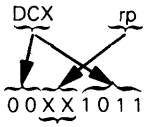
subtracts 1 from the contents of the memory byte with address $3714_{16}$. Status flags are modified as described for the DCR C instruction.

The DCR instruction is used in iterative instruction loops that use a counter with a value of 256 or less. This is the typical loop form:

```
        MVI     reg,data    ;LOAD INITIAL COUNTER VALUE
LOOP    -                   ;FIRST INSTRUCTION OF LOOP
        -
        -
        -
        DCR     reg         ;DECREMENT COUNTER
        JNZ     LOOP        ;RETURN IF NOT ZERO
```

## DCX — DECREMENT REGISTER PAIR




0 0 for rp = B, representing B,C
0 1 for rp = D, representing D,E
1 0 for rp = H, representing H,L
1 1 for rp = SP, representing the Stack Pointer

Subtract 1 from the 16-bit value contained in the specified register pair.

Suppose the Stack Pointer contains $2F7A_{16}$. After the instruction:

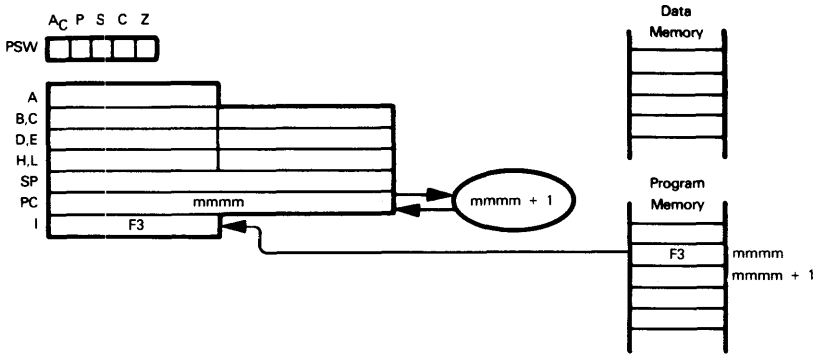
```
DCX     SP
```

has executed, the Stack Pointer will contain $2F79_{16}$.

The DCX instruction does not modify any status flags, and this is a defect in the 8080 instruction set. Whereas the DCR instruction is used in iterative instruction loops that use a counter with a value of 256 or less, the DCX instruction must be used if the counter value is more than 256. Since the DCX instruction sets no status flags, other instructions must be added simply to test for a zero result. This is a typical loop form:

```
        LXI     D,data16    ;LOAD INITIAL 16-BIT COUNTER VALUE
LOOP    -       -           ;FIRST INSTRUCTION OF LOOP
        -       -
        -       -
        -       -
        DCX     D           ;DECREMENT COUNTER
        MOV     A,D         ;TO TEST FOR ZERO, MOVE D TO A
        ORA     E           ;THEN OR A WITH E
        JNZ     LOOP        ;RETURN IF NOT ZERO
```

## DI — DISABLE INTERRUPTS

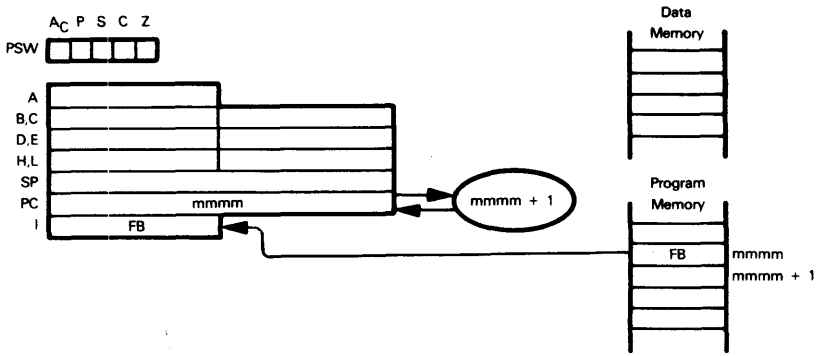



DI
F3

After this instruction has been executed the INTE signal is output low by the 8080 CPU, and no interrupt requests will be acknowledged. No registers or status flags are affected.

Remember that when an interrupt is acknowledged, interrupts are automatically disabled.

## EI — ENABLE INTERRUPTS




EI
FB

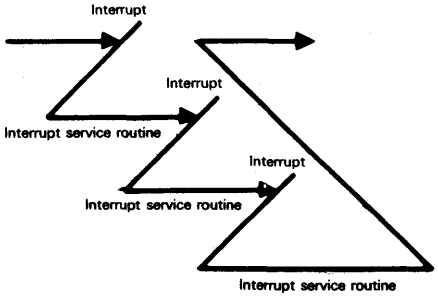
When this instruction is executed, interrupts are enabled, but not until one more instruction executes.

Most interrupt service routines end with the two instructions:

```
EI      ;ENABLE INTERRUPTS
RET     ;RETURN TO INTERRUPTED PROGRAM
```

If interrupts are processed serially, then for the entire duration of the interrupt service routine all interrupts are disabled — which means that in a multi-interrupt application there is a significant possibility for one or more interrupts to be pending when any interrupt service routine completes execution.

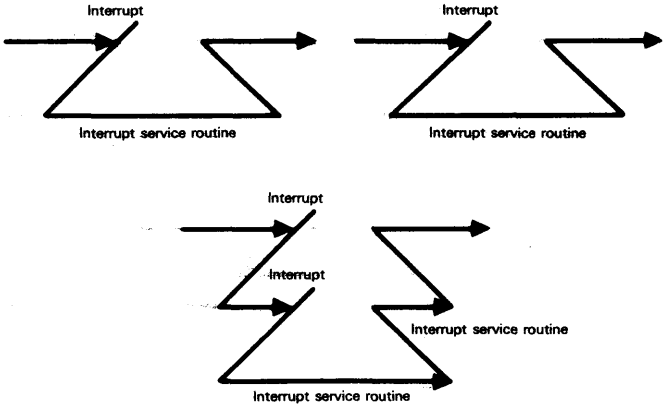
If interrupts were acknowledged as soon as the EI instructions had executed, then the Return instruction would not be executed. Under these circumstances returns would stack up one on top of the other — and unnecessarily consume stack memory space. This may be illustrated as follows:




By inhibiting interrupts for one more instruction following execution of EI, the 8080 CPU insures that the RET instruction gets executed in the sequence:

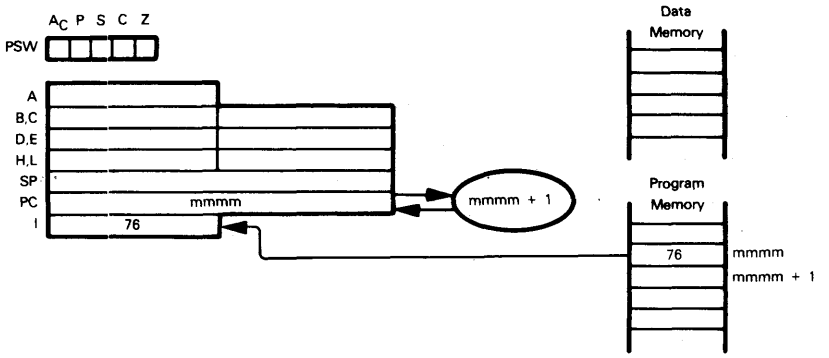
```
-
-
-
EI          ;ENABLE INTERRUPTS
RET         ;RETURN FROM INTERRUPT
```

It is not uncommon for interrupts to be kept disabled while an interrupt service routine is executing. Interrupts are processed serially:




If interrupts are processed serially, then priority arbitration will apply only during the acknowledge process.

## HLT — HALT

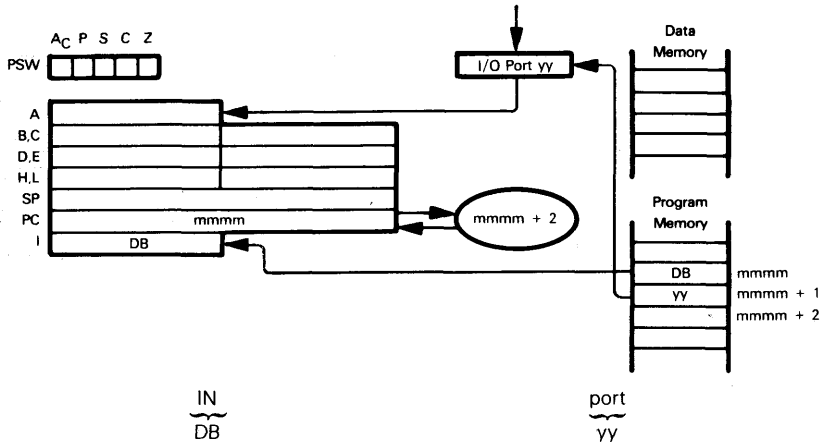



HLT
76

When the HLT instruction is executed, program execution ceases. It requires an interrupt or a reset to restart execution. No registers or statuses are affected.

**CAUTION: If interrupts are not enabled by an EI instruction prior to the HALT instruction, the 8080 CPU cannot exit the Halt state except by activation of the hardware Reset.**

## IN — INPUT TO ACCUMULATOR




Load a byte of data into the Accumulator from the I/O port identified by the second IN instruction object code byte.

Suppose $36_{16}$ is held in the buffer of I/O Port $1A_{16}$. After the instruction:

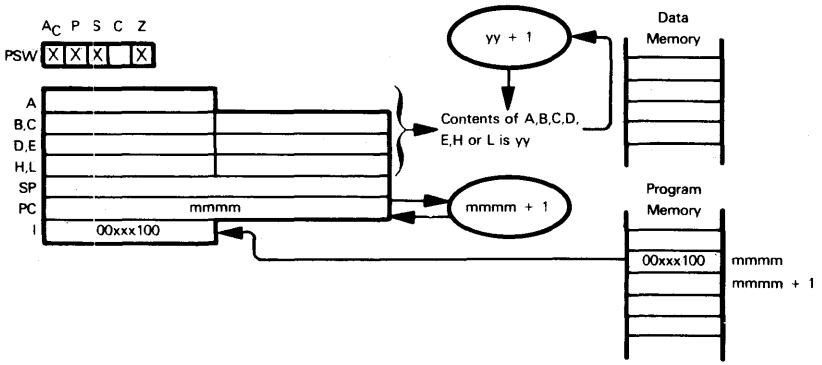
IN 1AH

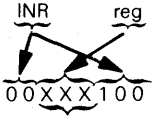
has executed, the Accumulator will contain $36_{16}$.

The IN instruction does not affect any statuses.

Use of the IN instruction is very hardware dependent. Valid I/O port addresses are determined by the way in which I/O logic has been implemented. It is also possible to design a microcomputer system that accesses external logic using memory reference instructions with specific memory addresses.

## INR — INCREMENT REGISTER OR MEMORY CONTENTS



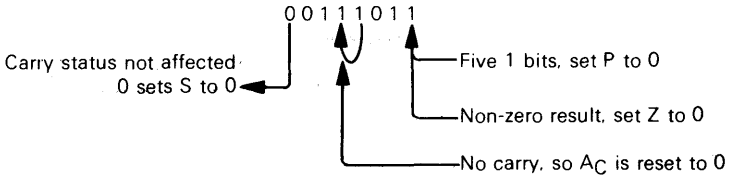

000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Add 1 to the contents of the specified register.

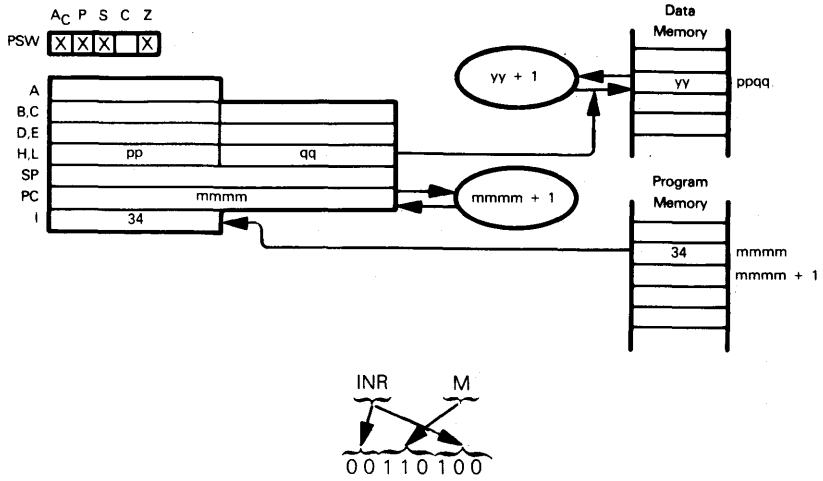
Suppose the C register contains $3A_{16}$. After the instruction:

INR C

has executed, the C register will contain $3B_{16}$:



The contents of a read/write memory byte may also be incremented:




Suppose HL contains $3714_{16}$. Then execution of the instruction:

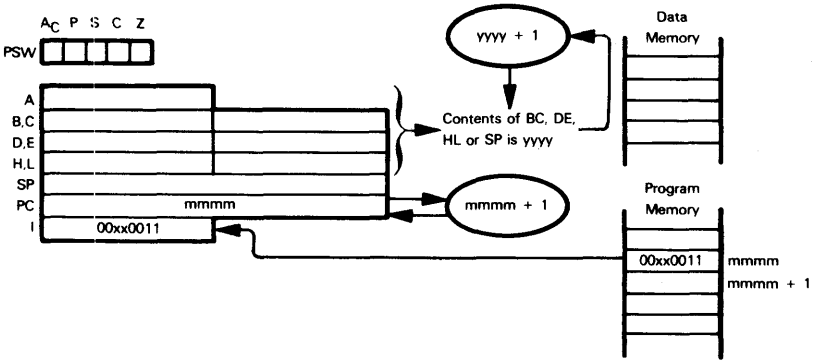
```
INR     M
```

adds 1 to the contents of the memory byte with address $3714_{16}$. Status flags are modified as described for the INR C instruction.

The INR instruction is used in iterative instruction loops that use a counter with a value of 256 or less. This is the typical loop form:

```
        MVI     reg,data    ;LOAD COMPLEMENT OF INITIAL COUNTER VALUE
LOOP    -       -           ;FIRST INSTRUCTION OF LOOP
        -       -
        -       -
        INR     reg         ;DECREMENT COUNTER
        JNZ     LOOP        ;RETURN IF NOT ZERO
```

## INX — INCREMENT REGISTER PAIR



0 0 for rp = B, representing B,C
0 1 for rp = D, representing D,E
1 0 for rp = H, representing H,L
1 1 for rp = SP, representing the Stack Pointer

Add 1 to the 16-bit value contained in the specified register pair.

Suppose the D and E registers contain $2F7A_{16}$. After the instruction:

```
INX     D
```

has executed, the D and E registers will contain $2F7B_{16}$.

The INX instruction does not modify any status flags, and this is a defect in the 8080 instruction set. Whereas the DCR instruction is used in iterative instruction loops that use a counter with a value of 256 or less, the INX instruction must be used if the counter value is more than 256. Since the INX instruction sets no status flags, other instructions must be added simply to test for a zero result. This is a typical loop form:

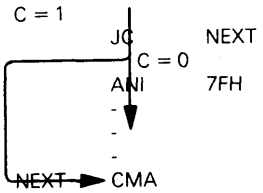
```
        LXI     D,data16    ;LOAD COMPLEMENT OF INITIAL 16-BIT
                            ;COUNTER VALUE
LOOP    -       -           ;FIRST INSTRUCTION OF LOOP
        -       -
        -       -
        -       -
        INX     D           ;INCREMENT COUNTER
        MOV     A,D         ;TO TEST FOR ZERO, MOVE D TO A,
        ORA     E           ;THEN OR A WITH E
        JNZ     LOOP        ;RETURN IF NOT ZERO
```

## JC — JUMP IF CARRY

| JC | label |
|---|---|
| DA | ppqq |

This instruction is identical to the JMP instruction except that the jump is only executed if the Carry status equals 1; otherwise, the next instruction is executed.

In the following instruction sequence:



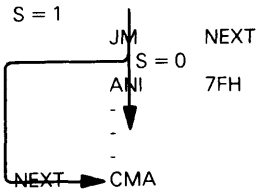

After the JC instruction, the CMA instruction is executed if the Carry status equals 1. The ANI instruction is executed if the Carry status equals 0.

## JM — JUMP IF MINUS

| JM | label |
|---|---|
| FA | ppqq |

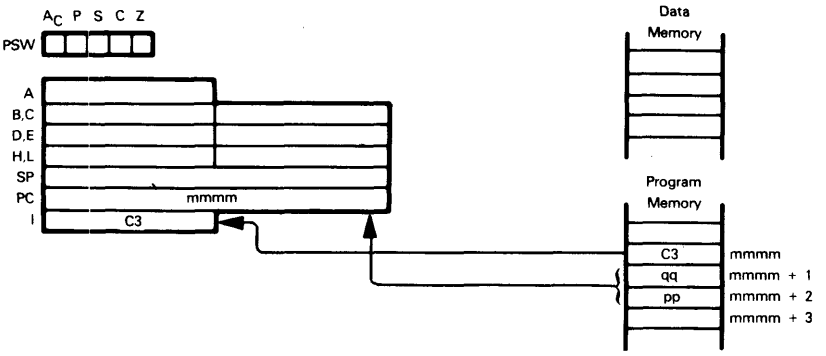
This instruction is identical to the JMP instruction except that the jump is only executed if the Sign status equals 1; otherwise, the next instruction is executed.

In the following instruction sequence:




After the JM instruction, the CMA instruction is executed if the Sign status equals 1. The ANI instruction is executed if the Sign status equals 0.

## JMP — JUMP TO THE INSTRUCTION IDENTIFIED IN THE OPERAND




JMP label

C3 ppqq

Load the contents of the Jump instruction object code second and third bytes into the Program Counter; this becomes the memory address for the next instruction to be executed. The previous Program Counter contents are lost.

In the following instruction sequence:

```
        JMP     NEXT
        ANI     7FH
        -
        -
        -
NEXT    CMA
```

After the JMP instruction, the CMA instruction will be executed. The ANI instruction will never be executed unless a Jump instruction somewhere else in the instruction sequence jumps to this instruction.

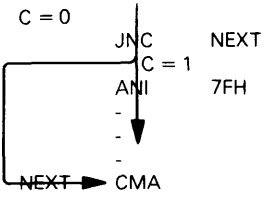
## JNC — JUMP IF NO CARRY

JNC label

D2 ppqq

This instruction is identical to the JMP instruction except that the jump is only executed if the Carry status equals 0; otherwise, the next instruction is executed.

In the following instruction sequence:



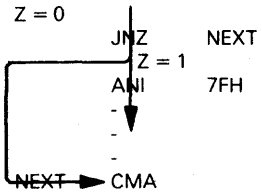

After the JNC instruction, the CMA instruction is executed if the Carry status equals 0. The ANI instruction is executed if the Carry status equals 1.

## JNZ — JUMP IF NOT ZERO

| JNZ | label |
|---|---|
| C2 | ppqq |

This instruction is identical to the JMP instruction except that the jump is only executed if the Zero status equals 0; otherwise, the next instruction is executed.

In the following instruction sequence:



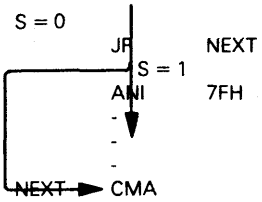

After the JNZ instruction, the CMA instruction is executed if the Zero status equals 0. The ANI instruction is executed if the Zero status equals 1.

## JP — JUMP IF PLUS

| JP | label |
|---|---|
| F2 | ppqq |

This instruction is identical to the JMP instruction except that the jump is only executed if the Sign status equals 0; otherwise, the next instruction is executed.

In the following instruction sequence:



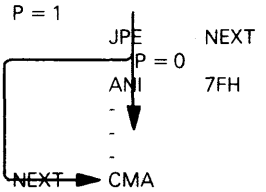

After the JP instruction, the CMA instruction is executed if the Sign status equals 0. The ANI instruction is executed if the Sign status equals 1.

## JPE — JUMP IF PARITY EVEN

| JPE | label |
|---|---|
| EA | ppqq |

This instruction is identical to the JMP instruction except that the jump is only executed if the Parity status equals 1; otherwise, the next instruction is executed.

In the following instruction sequence:



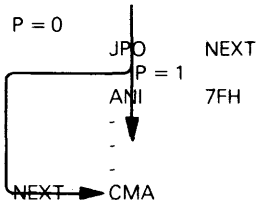

After the JPE instruction, the CMA instruction is executed if the Parity status equals 1. The ANI instruction is executed if the Parity status equals 0.

## JPO — JUMP IF PARITY ODD

| JPO | label |
|---|---|
| E2 | ppqq |

This instruction is identical to the JMP instruction except that the jump is only executed if the Parity status equals 0; otherwise, the next instruction is executed.

In the following instruction sequence:



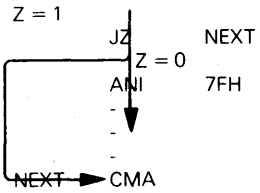

After the JPO instruction, the CMA instruction is executed if the Parity status equals 0. The ANI instruction is executed if the Parity status equals 1.

## JZ — JUMP IF ZERO

| JZ | label |
|---|---|
| CA | ppqq |

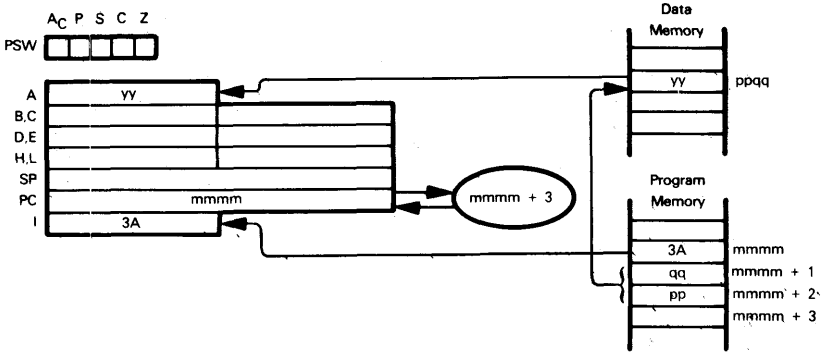
This instruction is identical to the JMP instruction except that the jump is only executed if the Zero status equals 1; otherwise, the next instruction is executed.

In the following instruction sequence:




After the JZ instruction, the CMA instruction is executed if the Zero status equals 1. The ANI instruction is executed if the Zero status equals 0.

## LDA — LOAD ACCUMULATOR FROM MEMORY USING DIRECT ADDRESSING




LDA addr

3A ppqq

Load into the Accumulator the contents of the memory byte addressed directly by the second and third bytes of the LDA instruction object code.

Suppose memory byte $084A_{16}$ contains $3A_{16}$. After the instruction:

```
LABEL   EQU     084AH
        -
        -
        -
        LDA     LABEL
```

has executed, the Accumulator will contain $3A_{16}$.

Remember that EQU is an Assembler Directive and not an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ wherever LABEL appears.

The instruction:

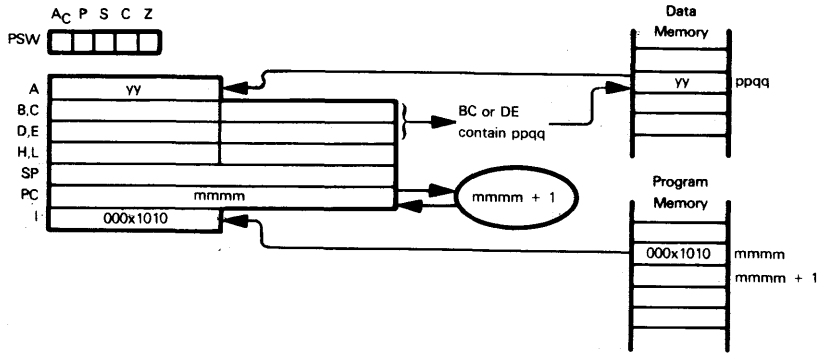
```
        LDA     LABEL
```

is equivalent to the two instructions:

```
        LXI     H,LABEL
        MOV     A,M
```

When you are loading a single data value from memory, the LDA instruction is preferred; it uses one instruction and three object program bytes to do what the LXI MOV combination does in two instructions and four object program bytes. Also, the LXI MOV combination uses the H and L registers; the LDA instruction does not.

## LDAX — LOAD ACCUMULATOR FROM MEMORY LOCATION ADDRESSED BY REGISTER PAIR



0 if rp is B, representing B,C
1 if rp is D, representing D,E

Load into the Accumulator the contents of the memory byte addressed by the BC or DE register pair.

Suppose the B register contains $08_{16}$, the C register contains $4A_{16}$ and memory byte $084A_{16}$ contains $3A_{16}$. After the instruction:

```
LDAX    B
```

has executed, the Accumulator will contain $3A_{16}$.

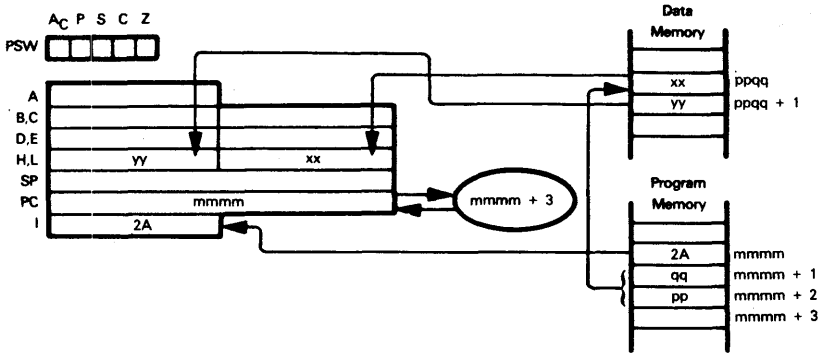
Note that there is no LDAX H instruction since this is identical to a MOV A,M instruction.

The LDAX and LXI instructions will normally be used together, since the LXI instruction loads a 16-bit address into the BC or DE registers as follows:

```
LXI     B,084AH
LDAX    B
```

Notice that the LDAX instruction will only load data into the Accumulator, whereas the MOV instruction will load data into any register.

## LHLD — LOAD H AND L REGISTERS DIRECT




The second and third object code bytes provide the memory address of a data byte, the contents of which will be loaded into the L register. The contents of the next sequential data byte is loaded into the H register.

Suppose memory byte $084A_{16}$ contains $3A_{16}$ and memory byte $084B_{16}$ contains $2C_{16}$. After the instruction:

```
LABEL   EQU     084AH
        -
        -
        -
        LHLD    LABEL
```

has executed, the H register will contain $2C_{16}$ and the L register will contain $3A_{16}$.

Remember that EQU is an assembler directive and not an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ wherever LABEL appears.

The LHLD instruction is a direct addressing version of the LXI H,DATA instruction. For example, the instruction:

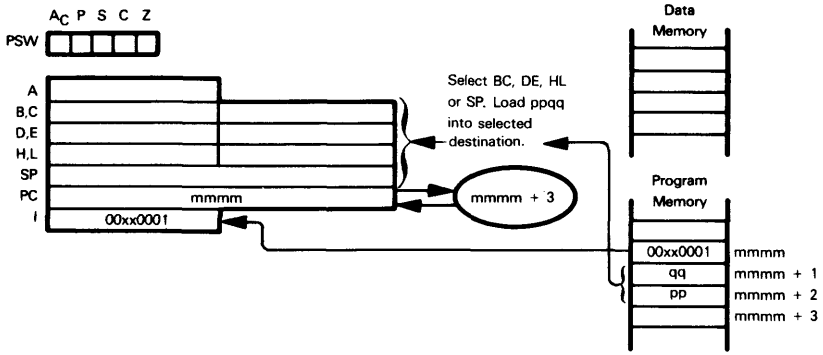
```
        LXI     H,2C3AH
```

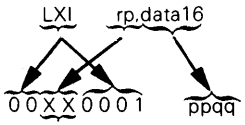
will also load $2C_{16}$ into the H register and $3A_{16}$ into the L register.

For 16-bit data that never changes, use LXI H,DATA instead of LHLD ADDR.

Remember, if ADDR directly addresses a byte of read/write memory, you can change the value that will be loaded into the H and L registers by an LHLD instruction. To do this, simply write the new value into ADDR and ADDR + 1.

## LXI — LOAD A 16-BIT VALUE, IMMEDIATE, INTO A REGISTER PAIR




0 0 if rp = B, selecting B and C registers
0 1 if rp = D, selecting D and E registers
1 0 if rp = H, selecting H and L registers
1 1 if rp = SP, selecting the Stack Pointer

Load into the selected register pair the contents of the second and third object code bytes.

After the instruction:

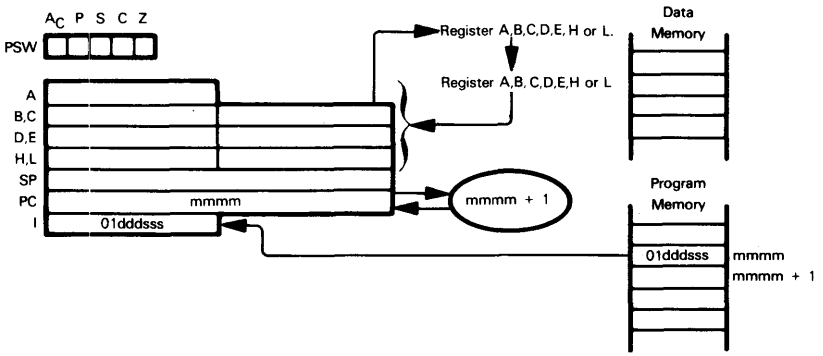
LXI SP,217AH

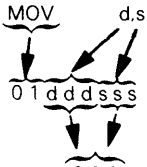
has executed, the Stack Pointer will contain $217A_{16}$.

LXI is the instruction most frequently used to load addresses into a register pair.

## MOV — MOVE DATA

This instruction takes two forms. Consider first the contents of one register being moved to another register.




000 d or s is Register B
001 d or s is Register C
010 d or s is Register D
011 d or s is Register E
100 d or s is Register H
101 d or s is Register L
111 d or s is the Accumulator

Move the contents of any register to any register. For example:

```
MOV     A,B
```

moves the contents of the B register to the Accumulator.

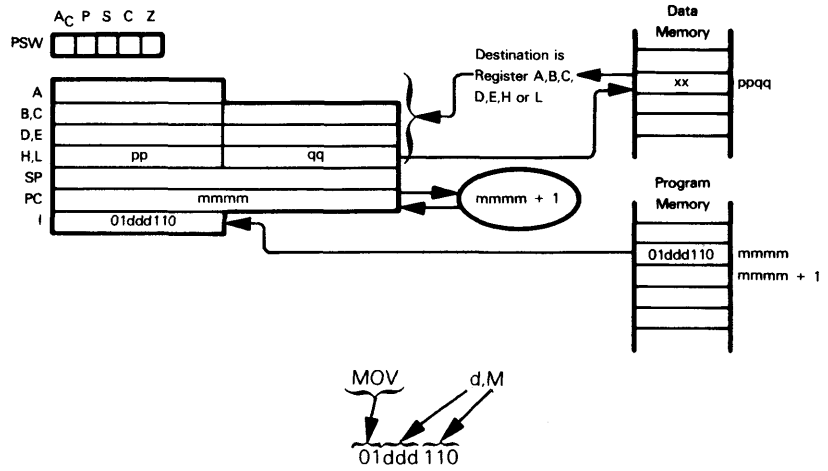
```
MOV     L,D
```

moves the contents of the D register to the L register.

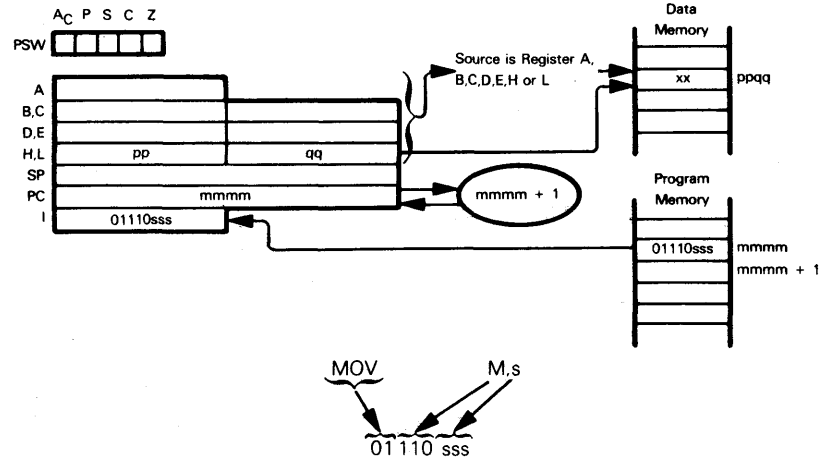
```
MOV     C,C
```

does nothing, since the C register has been specified as both the source and the destination.

A memory byte may also be the data source:




Or a memory byte may be the data destination:




In either case, ddd or sss is interpreted as for a register-to-register move.

Thus the instruction:

```
MOV     M,A
```

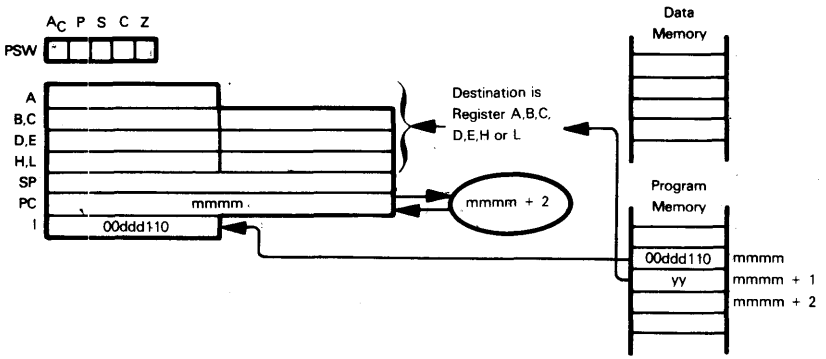
moves the Accumulator contents to the read/write memory byte addressed by the H and L registers. The instruction:

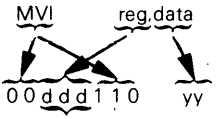
```
MOV     L,M
```

moves the contents of the memory byte addressed by the H and L registers into the L register.

The Move instruction in its various forms is the most frequently used of the 8080 instructions.

## MVI — LOAD DATA IMMEDIATE INTO REGISTER OR MEMORY




000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Move the contents of the second object code byte to one of the registers.

When the instruction:

```
MVI     A,2AH
```

has executed, $2A_{16}$ is loaded into the Accumulator. The instruction:

```
MVI     H,03H
```

loads $03_{16}$ into the H register.

Load Data Immediate Into Register instructions are very frequently used in 8080 programs.

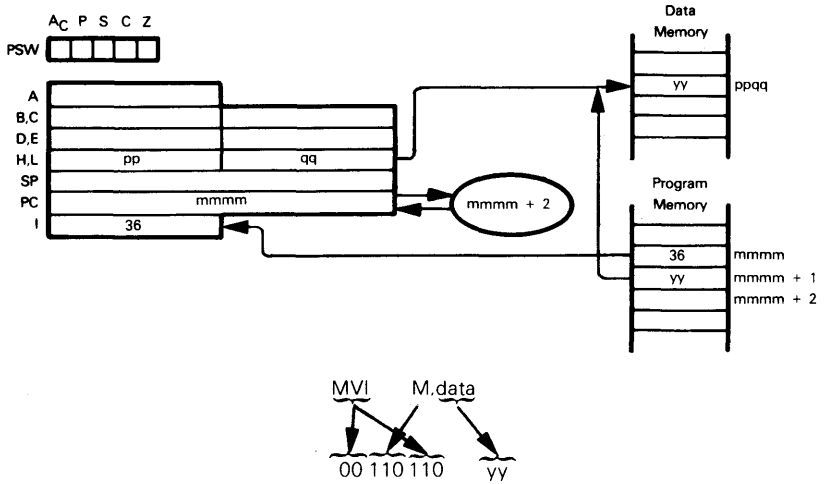
Notice that the LXI instruction is equivalent to two MVI instructions; for example:

```
LXI     H,032AH
```

is equivalent to:

```
MVI     H,03H
MVI     L,2AH
```

Data may also be loaded immediately into a byte of read/write memory:



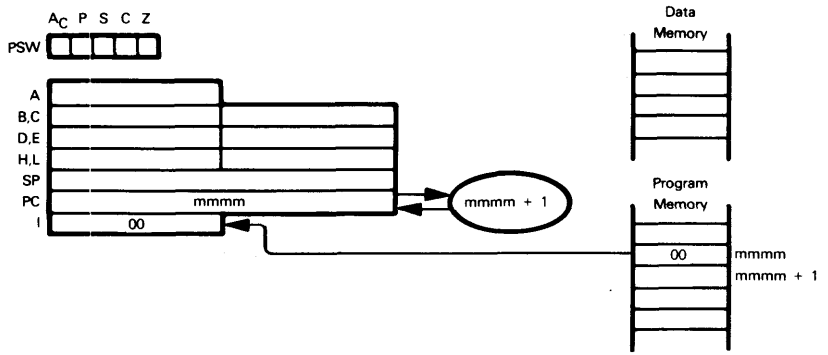

Suppose the H register contains $03_{16}$ and the L register contains $2A_{16}$; then when the instruction:

MVI M,2CH

has executed, $2C_{16}$ will be loaded into memory byte $032A_{16}$.

The Load Immediate Into Memory instruction (MVI M,data) is used much less than the Load Immediate Into Register instruction (MVI reg,data).

## NOP — NO OPERATION




NOP
00

Nothing happens when this instruction is executed; it is present for three reasons:

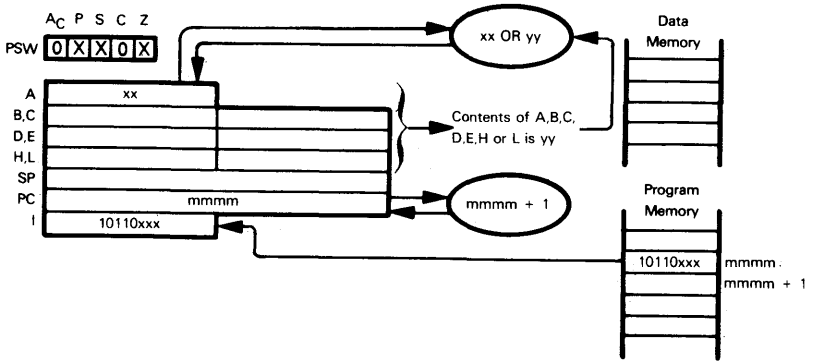
1) A program error that fetches an object code from non-existent memory will fetch 00. It is a good idea to insure that the commonest program error will do nothing.
2) The NOP instruction allows you to give a label to an object program byte:

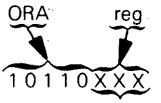
   HERE NOP

3) To fine-tune delay times. Each NOP instruction adds four clock cycles to a delay.

NOP is not a very useful or frequently used instruction.

## ORA — OR REGISTER OR MEMORY WITH ACCUMULATOR



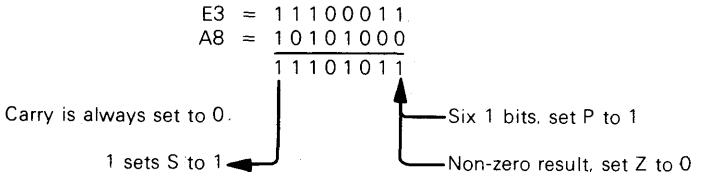

000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Logically OR the contents of the Accumulator with the contents of any register. Store the result in the Accumulator.

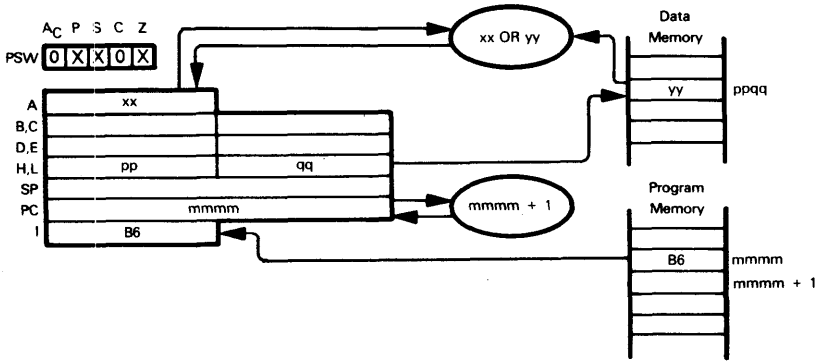
Suppose xx = $E3_{16}$, register E contains $A8_{16}$. After the instruction:

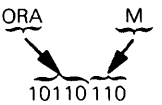
ORA E

has executed, the Accumulator will contain $EB_{16}$.



The contents of a memory byte may also be ORed with the Accumulator:




If xx = $E3_{16}$ and yy = $A8_{16}$, then execution of the instruction:

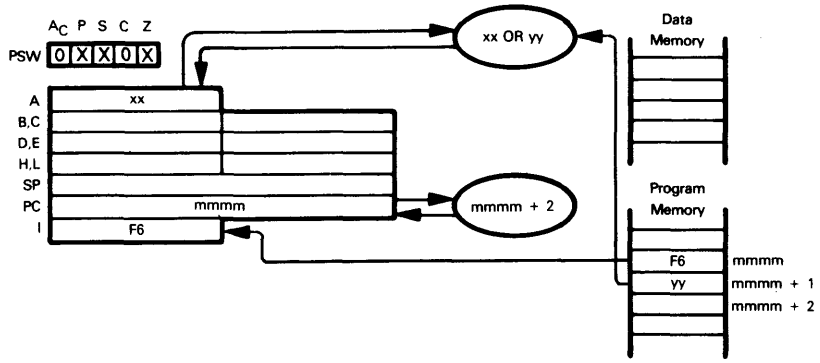
ORA M

generates the same result as execution of the ORA E instruction, which was just described.

ORA is not used as frequently as the OR immediate (ORI) instruction.

Note that ORing the Accumulator with itself (ORA A) allows you to clear the Carry status; this instruction is also used to set statuses following INX and DCX instructions.

## ORI — OR IMMEDIATE WITH ACCUMULATOR




ORI F6 data yy

OR the Accumulator with the contents of the second instruction object code byte.

Suppose xx = $3A_{16}$. After the instruction:

```
ORI     7CH
```

has executed, the Accumulator will contain $7E_{16}$:

```
3A = 0 0 1 1 1 0 1 0
7C = 0 1 1 1 1 1 0 0
     0 1 1 1 1 1 1 0
```

Carry is always set to 0

0 sets S to 0

Six 1 bits, set P to 1

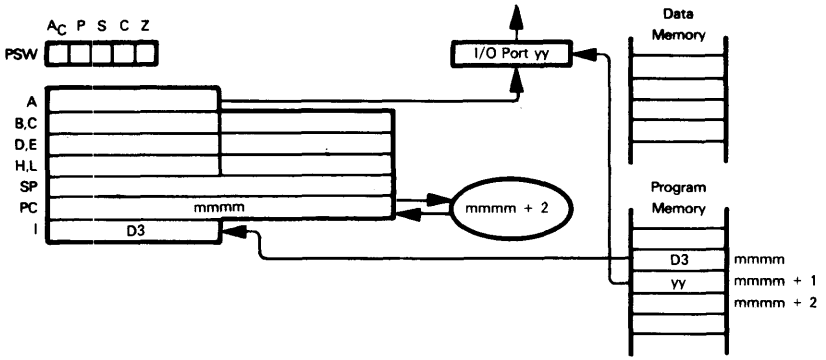
Non-zero result, set Z to 0

This is a routine logical instruction; it is often used to turn bits "on". For example, the instruction:

```
ORI     80H
```

will unconditionally set the high order Accumulator bit to 1.

## OUT — OUTPUT FROM ACCUMULATOR




OUT D3 port yy

Output the contents of the Accumulator to the I/O port identified by the second OUT instruction object code byte.

Suppose $36_{16}$ is held in the Accumulator. After the instruction:

OUT 1AH

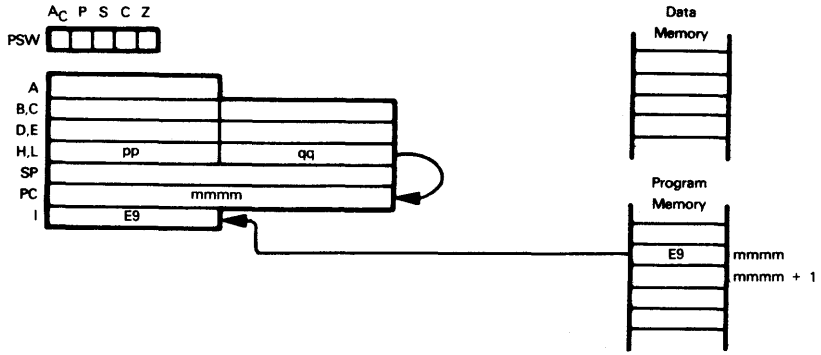
has executed, $36_{16}$ will be in the buffer of I/O Port $1A_{16}$.

The OUT instruction does not affect any statuses.

Use of the OUT instruction is very hardware dependent. Valid I/O port addresses are determined by the way in which I/O logic has been implemented. It is also possible to design a microcomputer system that accesses external logic using memory reference instructions with specific memory addresses.

OUT instructions are frequently used in special ways to control microcomputer logic external to the CPU.

## PCHL — JUMP TO ADDRESS SPECIFIED BY HL




PCHL
E9

The contents of the H and L registers are moved to the Program Counter; therefore an implied addressing jump is performed.

The instruction sequence:

```
LXI     H,ADDR
PCHL
```

has exactly the same net effect as the single instruction:

```
JMP     ADDR
```

Both specify that the instruction with label ADDR is to be executed next.

The PCHL instruction is useful when you want to increment a return address for a subroutine that has multiple returns.

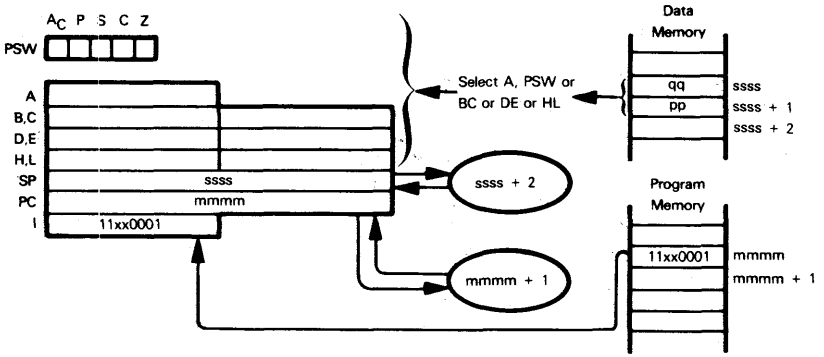
Consider the following call to subroutine SUB:

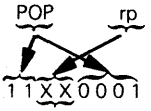
```
CALL    SUB     ;CALL SUBROUTINE
JMP     ERR     ;ERROR RETURN
                ;GOOD RETURN
```

Using RET to return from SUB would return execution to JMP ERR; therefore, if SUB executes without detecting error conditions, return as follows:

```
POP     H       ;POP RETURN ADDRESS TO HL
INX     H       ;ADD 3 TO RETURN ADDRESS
INX     H
INX     H
PCHL            ;RETURN
```

## POP — READ FROM THE TOP OF THE STACK




0 0 if rp is B, selecting B and C registers
0 1 if rp is D, selecting D and E registers
1 0 if rp is H, selecting H and L registers
1 1 if rp is PSW, selecting the Accumulator and the status flags as a 16-bit unit

Pop the two top stack bytes into the designated register pair.

Suppose qq = $03_{16}$ and pp = $2A_{16}$. Execution of the instruction:

POP H

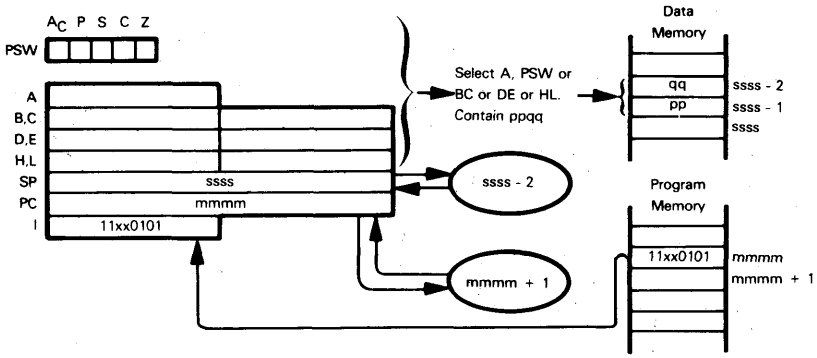
loads $03_{16}$ into the L register and $2A_{16}$ into the H register. Execution of the instruction:

POP PSW

loads $03_{16}$ into the status flags and $2A_{16}$ into the Accumulator. Thus the C status will be set to 1; other statuses will be cleared.

The POP instruction is most frequently used to restore register and status contents which have been saved on the stack, for example, while servicing an interrupt.

## PUSH — WRITE TO THE TOP OF THE STACK



0 0 if rp is B, selecting B and C registers
0 1 if rp is D, selecting D and E registers
1 0 if rp is H, selecting H and L registers
1 1 if rp is PSW, selecting the Accumulator and the status flags as a 16-bit unit

Push the contents of the designated register pair onto the top of the stack.

Suppose the H register contains $03_{16}$ and the L register contains $2A_{16}$. Execution of the instruction:

PUSH H

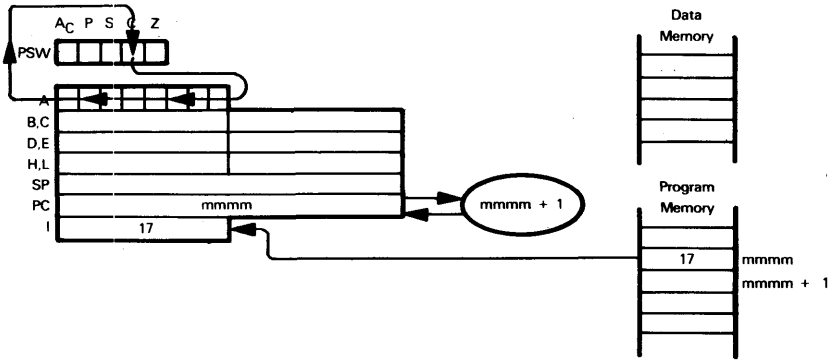
loads $03_{16}$ and then $2A_{16}$ into the top of the stack. Execution of the instruction:

PUSH PSW

loads the Accumulator and then the status flags into the top of the stack.

The PUSH instruction is most frequently used to save register and status contents, for example, before servicing an interrupt.

## RAL — ROTATE ACCUMULATOR LEFT THROUGH CARRY




RAL
17

Rotate Accumulator contents left one bit through Carry status.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the

RAL

instruction has executed, the Accumulator will contain $F5_{16}$ and the Carry status will be reset to 0:

| Accumulator | C | | Accumulator | C |
|---|---|---|---|---|
| 01111010 | 1 | → | 11110101 | 0 |

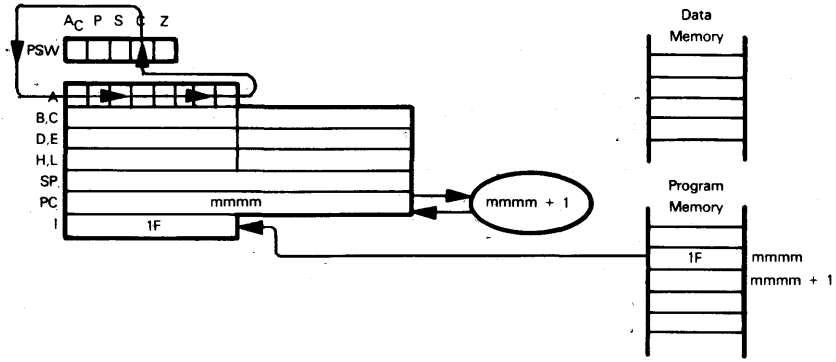
The RAL instruction is frequently used to perform multibyte left shifts, as described in An Introduction To Microcomputers, Volume I. The Carry status is cleared before performing the first left shift; subsequently the Carry status propagates the high-order bit of one byte into the low-order bit of the next byte. Here is an instruction sequence that shifts the contents of four memory bytes left, one bit:

```
        LXI     H,DATA      ;LOAD ADDRESS OF LOW ORDER DATA BYTE
        ANA     A           ;INITIALLY CLEAR CARRY
        MVI     B,3         ;USE REGISTER B AS A COUNTER
LOOP    MOV     A,M         ;LOAD DATA BYTE INTO ACCUMULATOR
        RAL                 ;ROTATE LEFT
        MOV     M,A         ;RESTORE RESULT
        INX     H           ;INCREMENT ADDRESS IN HL
        DCR     B           ;DECREMENT COUNTER
        JNZ     LOOP        ;RETURN FOR NEXT BYTE IF THERE IS ONE
```

**STATUS CONDITIONS**

Notice the careful thought that has been given to the statuses that are or are not set. RAL affects the Carry status only. INX and DCR affect the Zero, Sign and Parity statuses but not the Carry, which is therefore preserved from one execution of RAL to the next.

## RAR — ROTATE ACCUMULATOR RIGHT THROUGH CARRY




RAR
1F

Rotate Accumulator contents right one bit through Carry status.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the:

RAR

instruction has executed, the Accumulator will contain $BD_{16}$ and the Carry status will be reset to 0:

| Accumulator | C | | Accumulator | C |
|---|---|---|---|---|
| 01111010 | 1 | → | 10111101 | 0 |

The RAR instruction is frequently used to perform multibyte right shifts, as described in An Introduction To Microcomputers, Volume I. The Carry status is cleared before performing the first right shift; subsequently the Carry status propagates the low-order bit of one byte into the high-order bit of the next byte. Here is an instruction sequence that shifts the contents of four memory bytes right, one bit:

```
        LXI     H,DATA      ;LOAD ADDRESS OF LOW ORDER DATA BYTE
        ANA     A           ;INITIALLY CLEAR CARRY
        MVI     B,3         ;USE REGISTER B AS A COUNTER
LOOP    MOV     A,M         ;LOAD DATA BYTE INTO ACCUMULATOR
        RAR                 ;ROTATE RIGHT
        MOV     M,A         ;RESTORE RESULT
        INX     H           ;INCREMENT ADDRESS IN HL
        DCR     B           ;DECREMENT COUNTER
        JNZ     LOOP        ;RETURN FOR NEXT BYTE IF THERE IS ONE
```

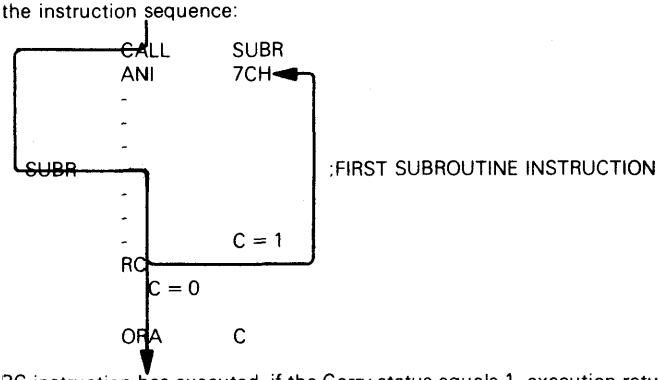
See the RAL description for a discussion of statuses.

## RC — RETURN IF THE CARRY STATUS EQUALS 1

RC
D8

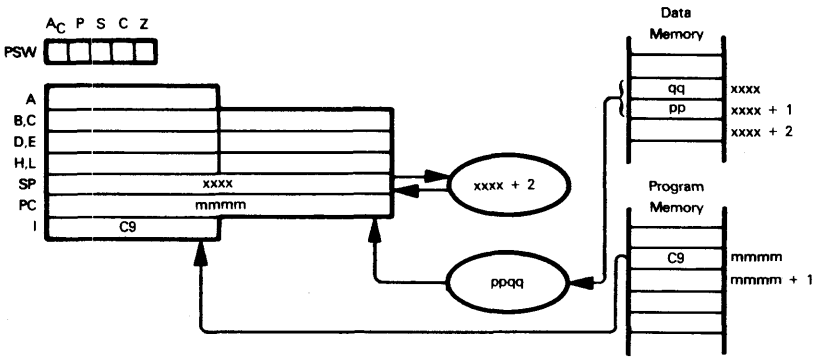
This instruction is identical to the RET instruction except that the return is not executed unless the Carry status equals 1 when the RC instruction is executed.

Consider the instruction sequence:




After the RC instruction has executed, if the Carry status equals 1, execution returns to the ANI instruction which follows the CALL. If the Carry status equals 0, the ORA instruction, being the next sequential instruction, is executed.

## RET — RETURN FROM SUBROUTINE



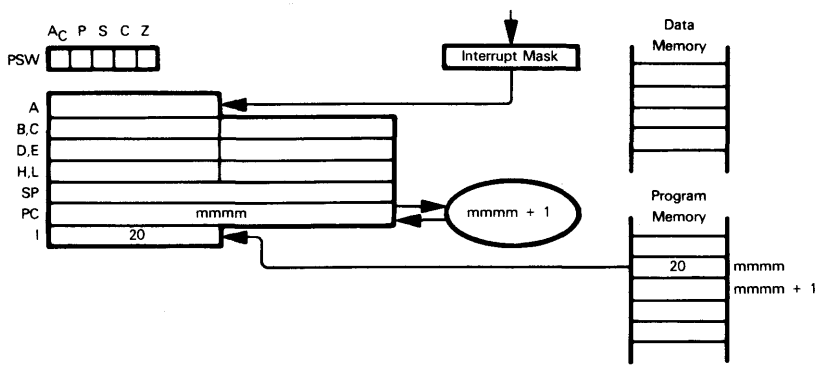

RET
C9

Move the contents of the top two stack bytes to the Program Counter; these two bytes provide the address of the next instruction to be executed. Previous Program Counter contents are lost. Increment the Stack Pointer by 2 to address the new top of stack.

Every subroutine must contain at least one Return (or conditional Return) instruction; this is the last instruction executed within the subroutine and causes execution to return to the calling program.

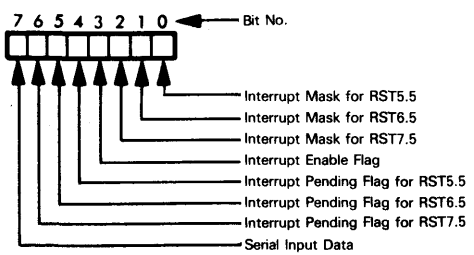
For an illustrated description of the RET instruction's execution, see Chapter 5.

## RIM — READ INTERRUPT MASK

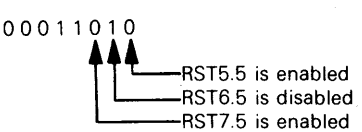



RIM
20

Load the contents of the restart interrupt mask and the serial input line into the Accumulator. The data loaded into the Accumulator is interpreted as follows:


7 6 5 4 3 2 1 0
Bit No.
Interrupt Mask for RST5.5
Interrupt Mask for RST6.5
Interrupt Mask for RST7.5
Interrupt Enable Flag
Interrupt Pending Flag for RST5.5
Interrupt Pending Flag for RST6.5
Interrupt Pending Flag for RST7.5
Serial Input Data

Bits 0, 1 and 2 are used to mask interrupts RST5.5, RST6.5 and RST7.5. If the bit corresponding to a particular RST is a 1, the RST is disabled. For example, if a RIM instruction results in $1A_{16}$ being loaded in the Accumulator, this is what happens:




RSTs 5.5, 6.5 and 7.5 work in a similar fashion to RSTs 0-7. When the 8085 detects one of these RSTs, it vectors as follows:

| Vectors | to | Location |
|---|---|---|
| RST5.5 | | $002C_{16}$ |
| RST6.5 | | $0034_{16}$ |
| RST7.5 | | $003C_{16}$ |

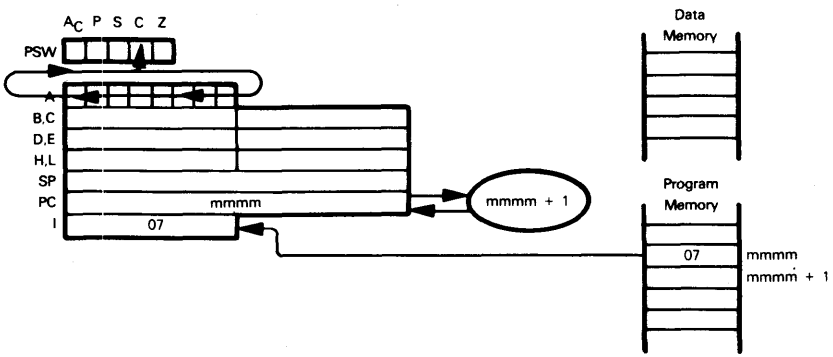
The other information in the interrupt mask is interpreted as follows:

Interrupt Enable Flag — If this flag is a 1, the interrupt system is enabled. If this flag is a 0, the interrupt system is disabled.

Interrupt Pending Flags — If this flag is a 1, an interrupt is being requested on the specified RST line. If this flag is a 0, no interrupt is being requested by the specified RST line.

Serial Input Data — This bit reflects the status of the SID line.

## RLC — ROTATE ACCUMULATOR LEFT




RLC
07

Rotate Accumulator contents left one bit.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the:

RLC

instruction has executed, the Accumulator will contain $F4_{16}$ and the Carry status will be reset to 0:

| Accumulator | C | | Accumulator | C |
|---|---|---|---|---|
| 01111010 | 1 | → | 11110100 | 0 |

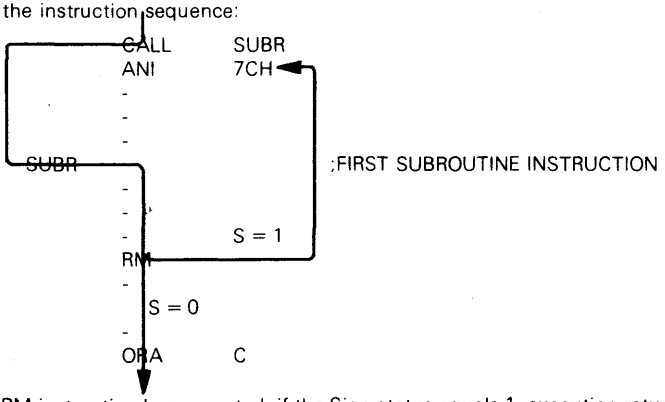
RLC should be used as a logical instruction.

## RM — RETURN IF THE SIGN STATUS EQUALS 1

RM
F8

This instruction is identical to the RET instruction, except that the return is not executed unless the Sign status equals 1 when the RM instruction is executed.

Consider the instruction sequence:



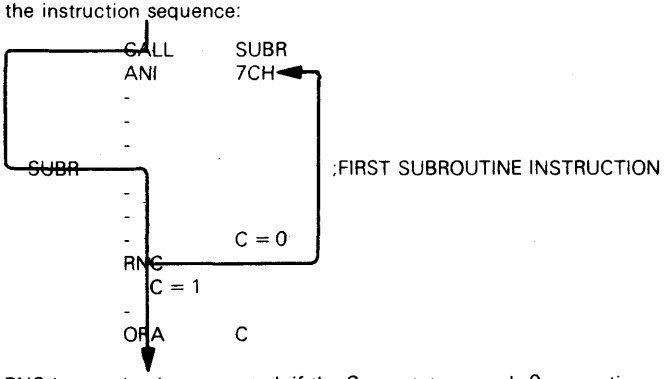

After the RM instruction has executed, if the Sign status equals 1, execution returns to the ANI instruction which follows the CALL. If the Sign status equals 0, the ORA instruction, being the next sequential instruction, is executed.

## RNC — RETURN IF THE CARRY STATUS EQUALS 0

RNC
D0

This instruction is identical to the RET instruction, except that the return is not executed unless the Carry status equals 0 when the RNC instruction is executed.

Consider the instruction sequence:



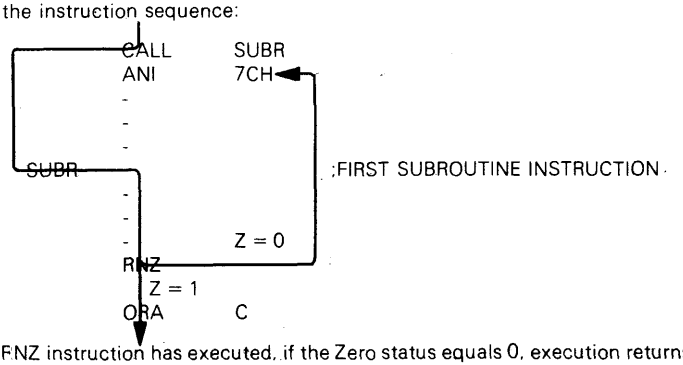

After the RNC instruction has executed, if the Carry status equals 0, execution returns to the ANI instruction which follows the CALL. If the Carry status equals 1, the ORA instruction, being the next sequential instruction, is executed.

## RNZ — RETURN IF THE ZERO STATUS EQUALS 0

RNZ
C0

This instruction is identical to the RET instruction, except that the return is not executed unless the Zero status equals 0 when the RNZ instruction is executed.

Consider the instruction sequence:



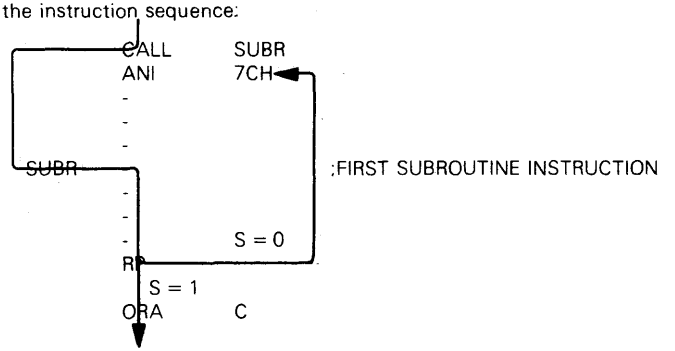

After the RNZ instruction has executed, if the Zero status equals 0, execution returns to the ANI instruction which follows the CALL. If the Zero status equals 1, the ORA instruction, being the next sequential instruction, is executed.

## RP — RETURN IF THE SIGN STATUS EQUALS 0

RP
F0

This instruction is identical to the RET instruction, except that the return is not executed unless the Sign status equals 0 when the RP instruction is executed.

Consider the instruction sequence:



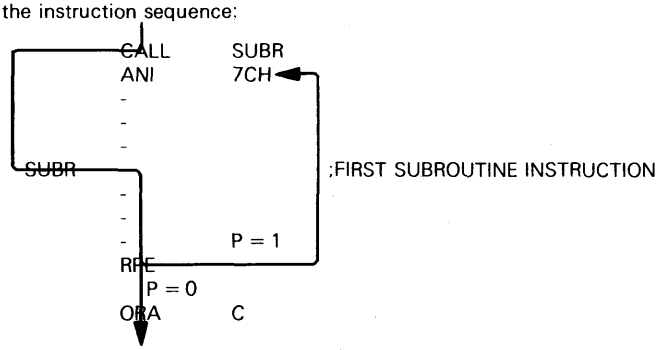

After the RP instruction has executed, if the Sign status equals 0, execution returns to the ANI instruction which follows the CALL. If the Sign status equals 1, the ORA instruction, being the next sequential instruction, is executed.

## RPE — RETURN IF THE PARITY STATUS EQUALS 1

RPE
E8

This instruction is identical to the RET instruction, except that the return is not executed unless the Parity status equals 1 when the RPE instruction is executed.

Consider the instruction sequence:



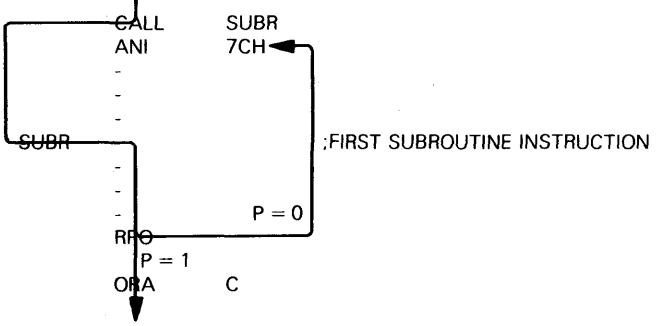

After the RPE instruction has executed, if the Parity status equals 1, execution returns to the ANI instruction which follows the CALL. If the Parity status equals 0, the ORA instruction, being the next sequential instruction, is executed.

## RPO — RETURN IF THE PARITY STATUS EQUALS 0

RPO
E0

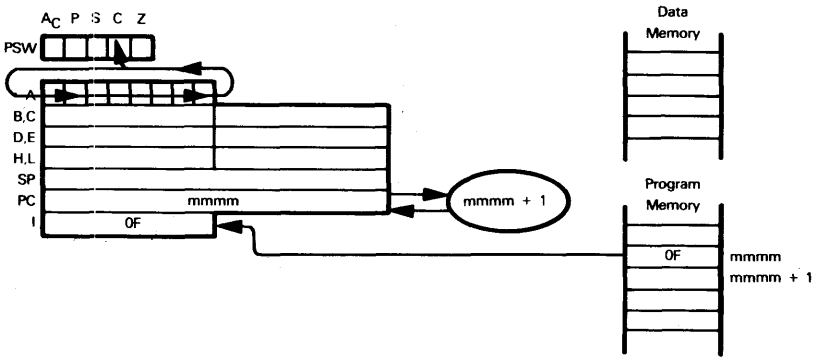
This instruction is identical to the RET instruction, except that the return is not executed unless the Parity status equals 0 when the RPO instruction is executed.

Consider the instruction sequence:




After the RPO instruction has executed, if the Parity status equals 0, execution returns to the ANI instruction which follows the CALL. If the Parity status equals 1, the ORA instruction, being the next sequential instruction, is executed.

## RRC — ROTATE ACCUMULATOR RIGHT




RRC
0F

Rotate Accumulator contents right one bit.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the:

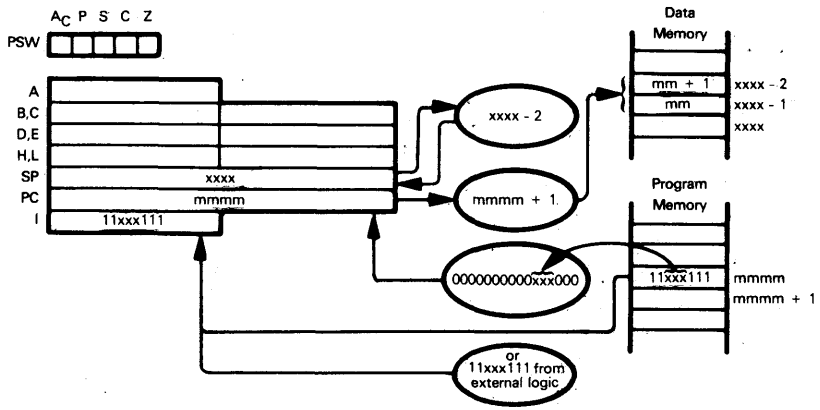
RRC

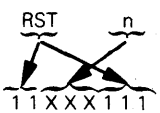
instruction has executed, the Accumulator will contain $3D_{16}$ and the Carry status will be reset to 0:

| Accumulator | C | | Accumulator | C |
|---|---|---|---|---|
| 01111010 | 1 | → | 00111101 | 0 |

RRC should be used as a logical instruction.

## RST — RESTART




Call the subroutine origined at the low memory address specified by n.

When the instruction:

RST 3

is executed, the subroutine origined at memory location $0018_{16}$ is called. The previous Program Counter contents are pushed to the top of the stack.

Usually the RST instruction is used in conjunction with interrupt processing, as described in Chapter 5.

**SUBROUTINE CALL USING RST**

**If your application does not use all RST instruction codes to service interrupts, do not overlook the possibility of calling subroutines using RST instructions.** Origin frequently used subroutines at appropriate RST addresses, and these subroutines can be called with a single-byte RST instruction instead of a three-byte CALL instruction.

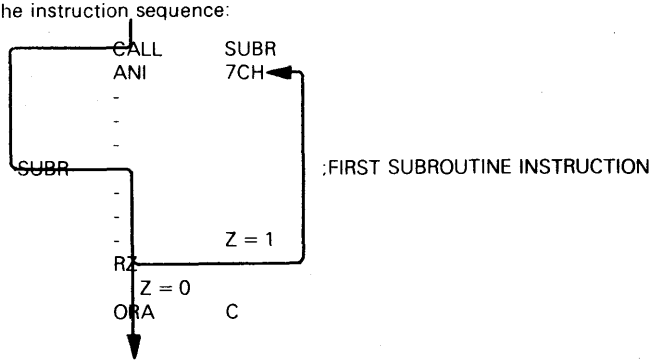
## RZ — RETURN IF THE ZERO STATUS EQUALS 1

RZ

C8

This instruction is identical to the RET instruction, except that the return is not executed unless the Zero status equals 1 when the RZ instruction is executed.

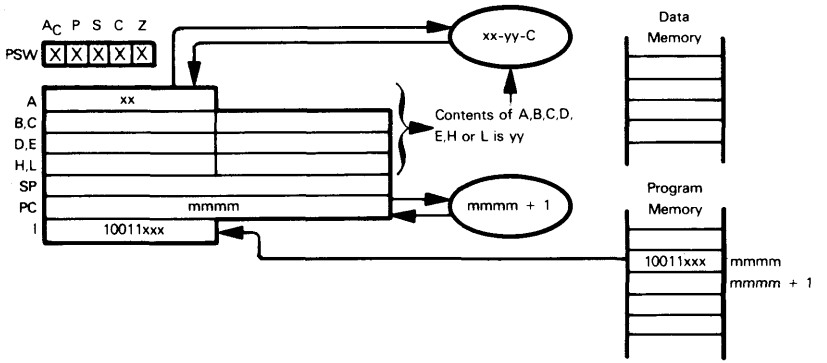
Consider the instruction sequence:

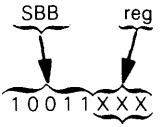



After the RZ instruction has executed, if the Zero status equals 1, execution returns to the ANI instruction which follows the CALL. If the Zero status equals 0, the ORA instruction, being the next sequential instruction, is executed.

## SBB — SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR WITH BORROW

This instruction takes two forms. First consider a register's contents subtracted from the Accumulator:



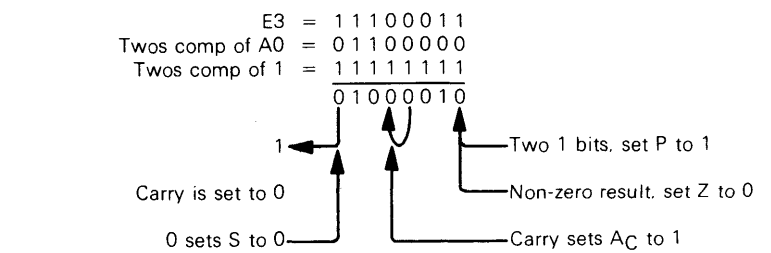

000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Subtract the contents of the specified register and the Carry status from the Accumulator, treating register contents as simple binary data.

Suppose xx = $E3_{16}$, register E contains $A0_{16}$, C = 1. After the instruction:

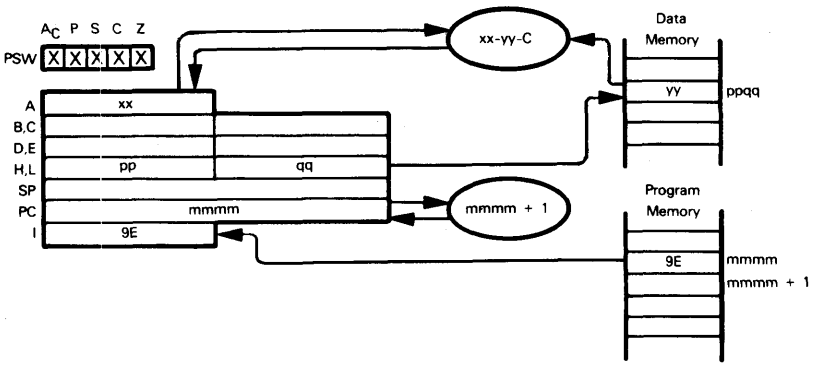
SBB E

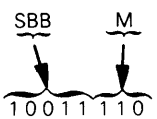
has executed, the Accumulator will contain $42_{16}$:


E3 = 11100011
Twos comp of A0 = 01100000
Twos comp of 1 = 11111111
01000010
1
Carry is set to 0
0 sets S to 0
Two 1 bits, set P to 1
Non-zero result, set Z to 0
Carry sets Ac to 1

Notice that the resulting Carry is complemented.

The contents of a memory byte may also be subtracted, with borrow, from the Accumulator:



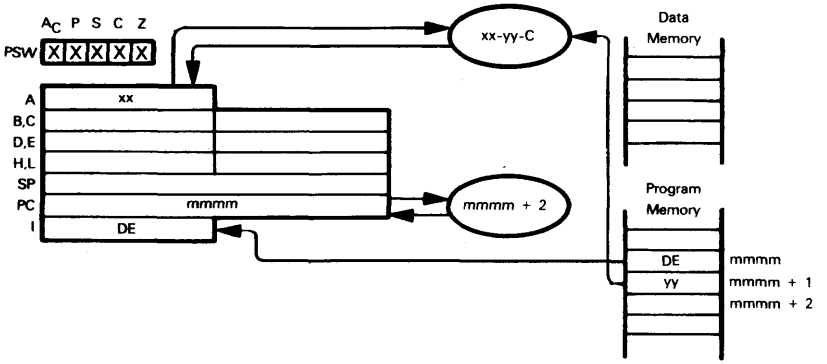

If xx = $E3_{16}$, yy = $A0_{16}$ and C = 1, then execution of the instruction:

```
SBB     M
```

generates the same result as execution of the SBB E instruction, which was just described.

The SBB instruction is used in multibyte subtraction after the low-order byte has been processed using the SUB instruction.

## SBI — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR WITH BORROW



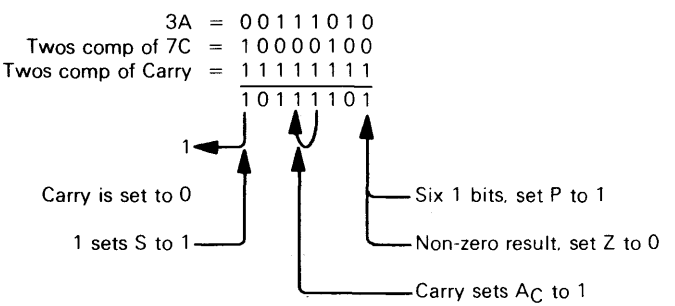

SBI (DE) data (yy)

Subtract the contents of the second instruction code byte and the Carry status from the Accumulator.

Suppose xx = $3A_{16}$ and the Carry status equals 1. After the instruction:

SBI 7CH

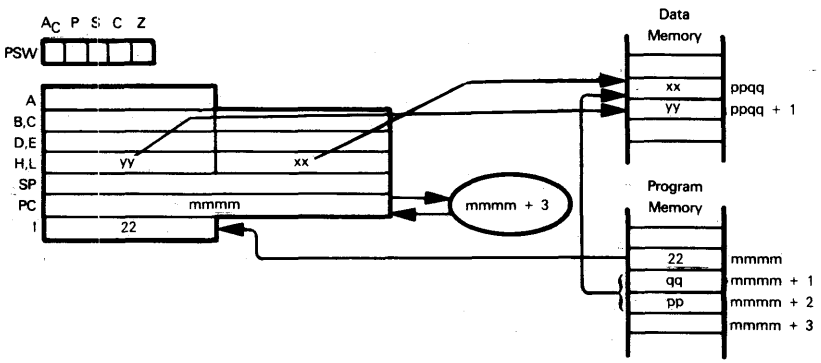
has executed, the Accumulator will contain $BD_{16}$:


3A = 0 0 1 1 1 0 1 0
Twos comp of 7C = 1 0 0 0 0 1 0 0
Twos comp of Carry = 1 1 1 1 1 1 1 1
1 0 1 1 1 1 0 1
1
Carry is set to 0
1 sets S to 1
Six 1 bits, set P to 1
Non-zero result, set Z to 0
Carry sets AC to 1

Notice that the resulting Carry is complemented.

This instruction is not used as frequently as SUI.

## SHLD — STORE H AND L REGISTERS DIRECT




SHLD addr
22 ppqq

The second and third object code bytes provide the memory address of a data byte into which the L register contents are written. The H register contents are written into the next sequential data byte.

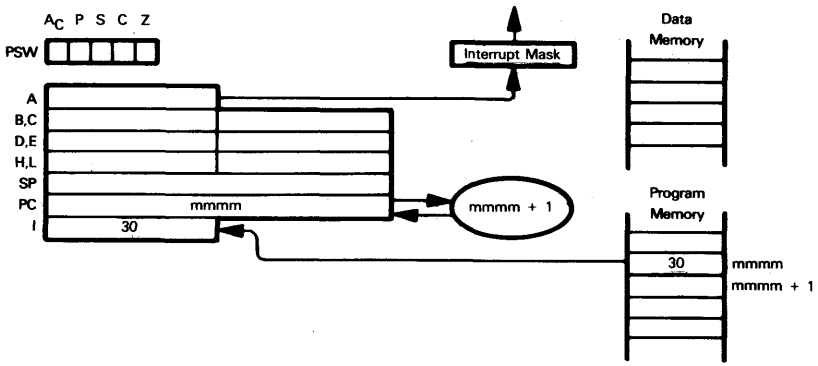
Suppose xx = $2C_{16}$ and yy = $3A_{16}$. After the instruction:

```
LABEL   EQU     084AH
        .
        .
        .
        SHLD    LABEL
```

has executed, memory byte $084A_{16}$ will contain $2C_{16}$. Memory byte $084B_{16}$ will contain $3A_{16}$.

Remember that EQU is an assembler directive and not an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ whenever LABEL appears.

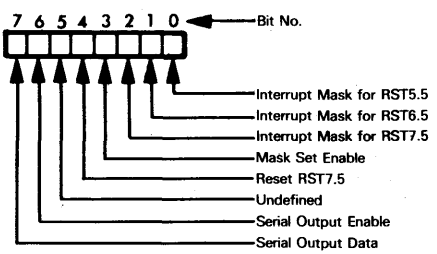
## SIM — SET INTERRUPT MASK




SIM
30

Output the contents of the Accumulator to the restart interrupt mask and the serial output line (SOD).

The contents of the Accumulator are interpreted in the following manner:



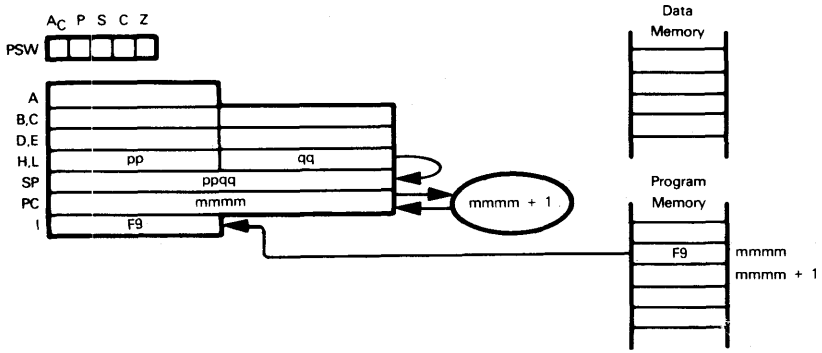

The interrupt mask function is only performed if the Mask Set Enable bit is 1. For example, if $0F_{16}$ is held in the Accumulator and the SIM instruction is performed, 1 will be sent to the interrupt mask for all three RSTs. If, however, $07_{16}$ was held in the Accumulator, the interrupt masks would not be affected.

If the interrupt mask for a particular RST is a 1, the RST is disabled. If the interrupt mask for a particular RST is a 0, the RST is enabled.

The reset RST7.5 bit is used to clear an internal request flip-flop for RST7.5 if the bit is 1.

The serial output data is only performed if the Serial Output Enable bit is 1. For example, if $03_{16}$ is held in the Accumulator and the SIM instruction is performed, the SOD latch will not be affected. If, however, $47_{16}$ is held in the Accumulator, a 0 will be loaded into the SOD latch if a SIM instruction is performed.

## SPHL — LOAD THE STACK POINTER FROM THE H AND L REGISTERS




SPHL
F9

Move the contents of the H and L registers to the Stack Pointer.

Suppose pp = $08_{16}$ and qq = $3F_{16}$. After the instruction:

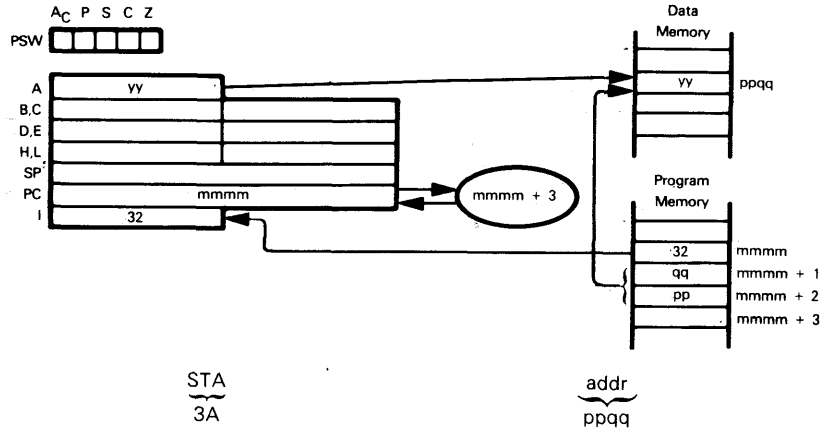
SPHL

has executed, the Stack Pointer will contain $083F_{16}$.

**The SPHL instruction can be used to access two stacks** — with an idle address preserved in the H and L registers. Stacks are frequently used in this fashion to access text strings or any such data that must be accessed byte-serially.

**The important point to bear in mind is that stack logic can be used instead of implied memory addressing with auto-increment.**

## STA — STORE ACCUMULATOR IN MEMORY USING DIRECT ADDRESSING




Store the Accumulator contents in the memory byte addressed directly by the second and third bytes of the STA instruction object code.

Suppose the Accumulator contains $3A_{16}$. After the instruction:

```
LABEL   EQU     084AH
        -
        -
        -
        STA     LABEL
```

has executed, memory byte $084A_{16}$ will contain $3A_{16}$.

Remember that EQU is an assembler directive and not an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ wherever LABEL appears.

The instruction:

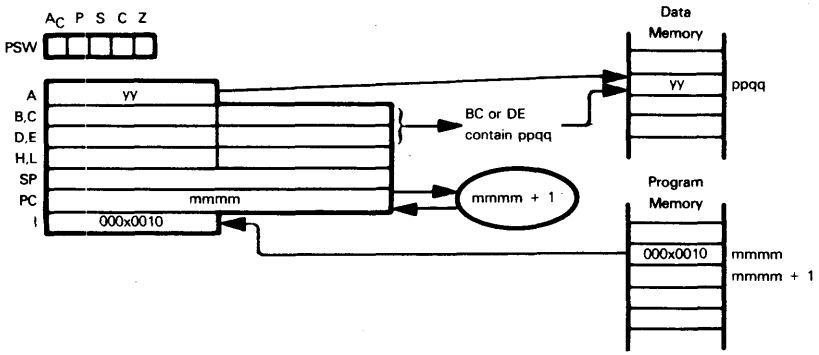
```
        STA     LABEL
```

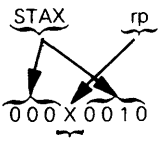
is equivalent to the two instructions:

```
        LXI     H,LABEL
        MOV     M,A
```

When you are storing a single data value in memory, the STA instruction is preferred; it uses one instruction and three object program bytes to do what the LXI MOV combination does in two instructions and four object program bytes. Also, the LXI MOV combination uses the H and L registers; the STA instruction does not.

## STAX — STORE ACCUMULATOR IN THE MEMORY LOCATION ADDRESSED BY A REGISTER PAIR




0 if rp is B, representing B,C
1 if rp is D, representing D,E

Store the Accumulator contents in the memory byte addressed by the BC or DE register pair.

Suppose the B register contains $08_{16}$, the C register contains $4A_{16}$ and the Accumulator contains $3A_{16}$. After the instruction:

```
STAX    B
```

has executed, memory byte $084A_{16}$ will contain $3A_{16}$.

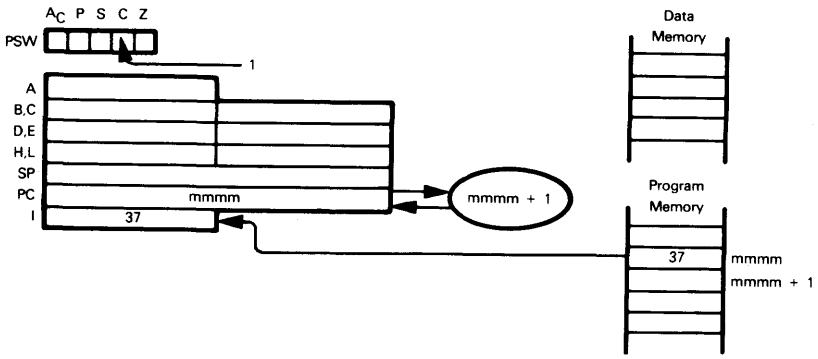
Note that there is no STAX H instruction, since this is identical to a MOV M,A instruction.

Normally the STAX and LXI instructions will be used together, since the LXI instruction loads a 16-bit address into the BC or DE registers, as follows:

```
LXI     B,084AH
STAX    B
```

Notice that the STAX instruction will only store data from the Accumulator, whereas the MOV instruction will store data from any register.

## STC — SET CARRY STATUS

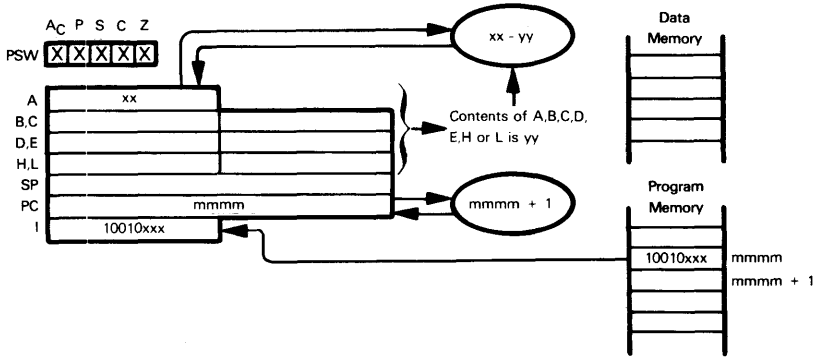


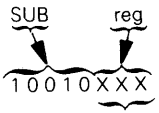

STC
37

When the STC instruction is executed the Carry status is set to 1, regardless of its previous value. No other statuses or register contents are affected.

## SUB — SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR

This instruction takes two forms. First consider a register's contents subtracted from the Accumulator:



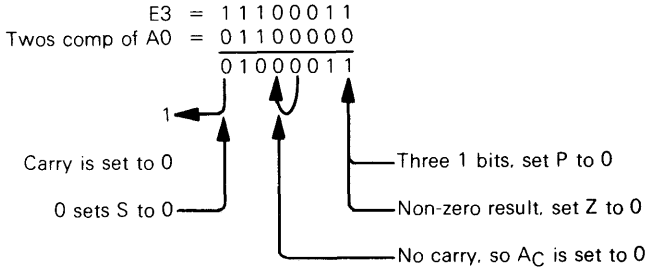

000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Subtract the contents of the specified register from the Accumulator, treating register contents as simple binary data.

Suppose xx = $E3_{16}$ and register E contains $A0_{16}$. After the instruction:

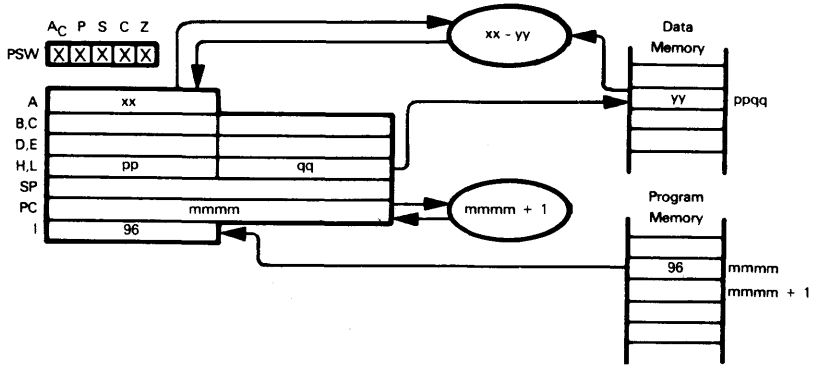
SUB E

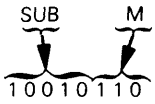
has executed, the Accumulator will contain $43_{16}$:


E3 = 11100011
Twos comp of A0 = 01100000
01000011
1
Carry is set to 0
0 sets S to 0
Three 1 bits, set P to 0
Non-zero result, set Z to 0
No carry, so Ac is set to 0

Notice that the resulting Carry is complemented.

The contents of a memory byte may also be subtracted from the Accumulator:



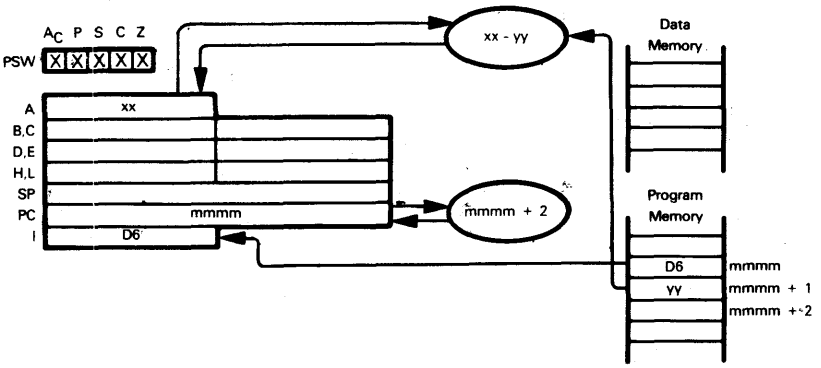

If xx = $E3_{16}$ and yy = $A0_{16}$, then execution of the instruction:

```
SUB     M
```

generates the same result as execution of the SUB E instruction, which was just described.

The SUB instruction is used to perform single-byte subtractions, or for the low-order byte in multibyte subtractions.

## SUI — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR




SUI (D6) data (yy)

Subtract the contents of the second instruction code byte from the Accumulator.

Suppose xx = $3A_{16}$. After the instruction:

SBI 7CH

has executed, the Accumulator will contain $BE_{16}$:

```
3A  =  00111010
Twos comp of 7C  =  10000100
                    10111110
```

No carry, so Carry is set to 1

1 sets S to 1

Six 1 bits, set P to 1

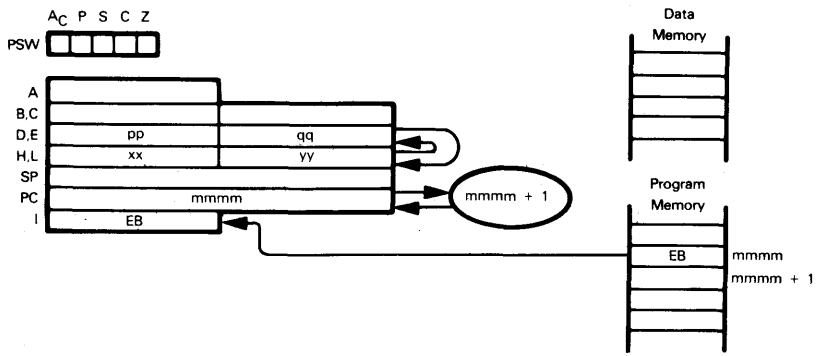
Non-zero result, set Z to 0

No carry, so $A_C$ is set to 0

Notice that the resulting Carry is complemented.

This instruction is the preferred Subtract Immediate.

## XCHG — EXCHANGE DE AND HL REGISTERS' CONTENTS




XCHG

EB

The D and E registers' contents are swapped with the H and L registers' contents.

Suppose pp $=03_{16}$, qq $= 2A_{16}$, xx $= 41_{16}$ and yy $= FC_{16}$. After the instruction:

XCHG

has executed, H will contain $03_{16}$, L will contain $2A_{16}$, D will contain $41_{16}$ and E will contain $FC_{16}$.

The two instructions:

```
XCHG
MOV     A,M
```

are equivalent to:

```
LDAX    D
```

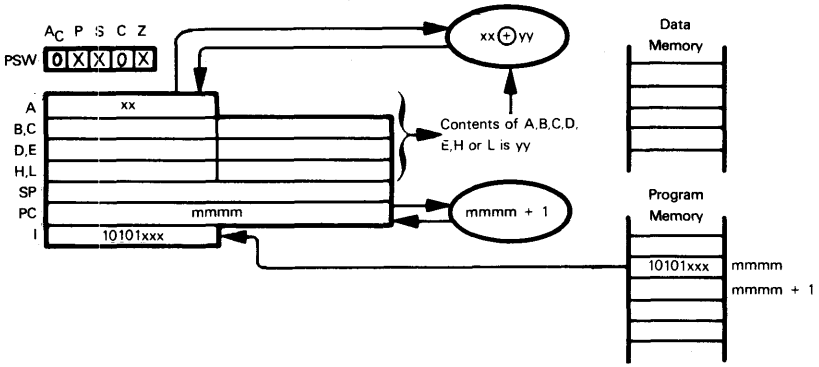
But if you want to load data addressed by the D and E registers into the B register,

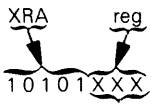
```
XCHG
MOV     B,M
```

has no single instruction equivalent.

## XRA — EXCLUSIVE-OR REGISTER OR MEMORY WITH ACCUMULATOR

This instruction takes two forms. First consider a register's contents Exclusive-ORed with the Accumulator:



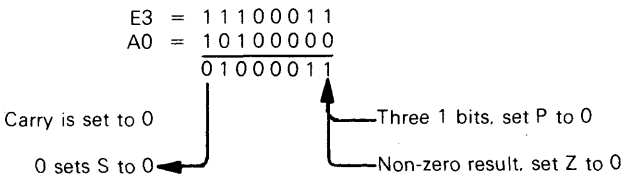

000 for reg = B
001 for reg = C
010 for reg = D
011 for reg = E
100 for reg = H
101 for reg = L
111 for reg = A

Exclusive-OR the contents of the specified register with the Accumulator, treating register contents as simple binary data.

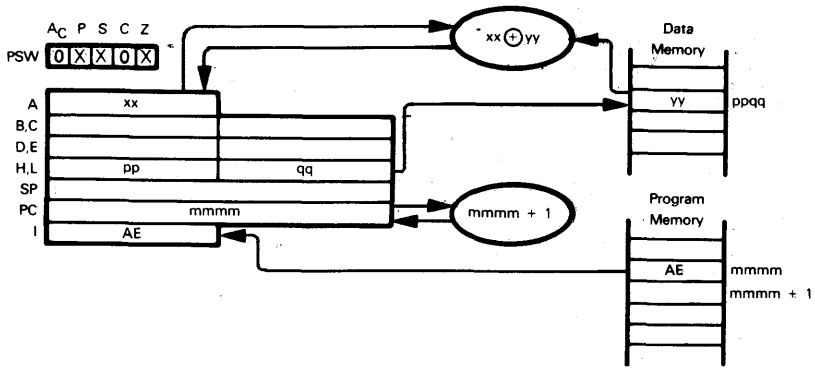
Suppose xx = $E3_{16}$ and register E contains $A0_{16}$. After the instruction:

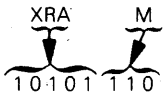
XRA E

has executed, the Accumulator will contain $43_{16}$:



The contents of a memory byte may also be Exclusive-ORed with the Accumulator:



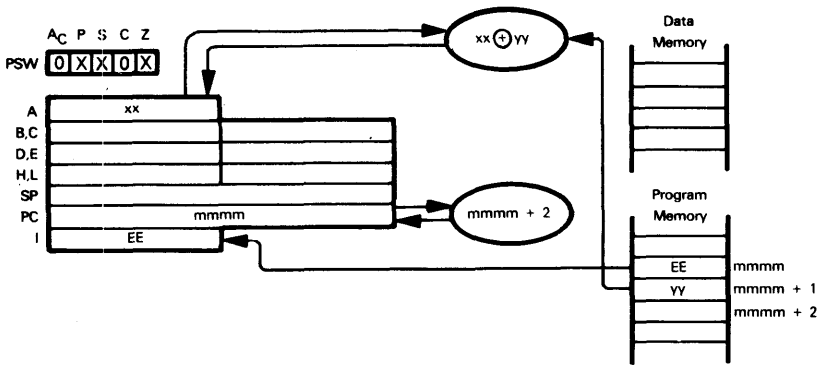

If xx = $E3_{16}$ and yy = $A0_{16}$, then execution of the instruction:

XRA M

generates the same result as execution of the XRA E instruction, which was just described.

The Exclusive-OR instruction is used to test for changes in bit status.

## XRI — EXCLUSIVE-OR IMMEDIATE DATA WITH ACCUMULATOR




XRI (EE) data (yy)

Exclusive-OR the contents of the second instruction code byte with the Accumulator.

Suppose xx = $3A_{16}$. After the instruction:

```
XRI     7CH
```

has executed, the Accumulator will contain $46_{16}$:

```
3A = 00111010
7C = 01111100
     01000110
```

Carry is set to 0

0 sets S to 0

Three 1 bits, set P to 1

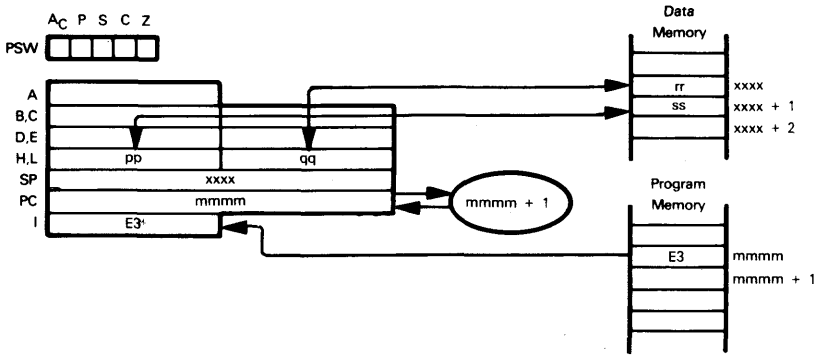
Non-zero result, set Z to 0

This is a routine logical instruction; it is often used to complement bits. For example, the instruction:

```
XRI     03H
```

will unconditionally complement the two low-order bits of the Accumulator.

## XTHL — EXCHANGE TOP OF STACK WITH HL




XTHL

E3

Exchange the contents of the L register with the top stack byte. Exchange the contents of the H register with the byte below the stack top.

Suppose pp = $21_{16}$, qq = $FA_{16}$, rr = $3A_{16}$, ss = $E2_{16}$. After the instruction:

XTHL

has executed, H will contain $E2_{16}$, L will contain $3A_{16}$ and the top two stack bytes will contain $FA_{16}$ and $21_{16}$, respectively.

The two instructions:

XTHL
XTHL

executed in sequence are equivalent to no operation.

The XTHL instruction is used to access and manipulate data at the top of the stack, as illustrated in the multiple subroutine discussion in Chapter 5.

# INTEL 8080A AND 8085 ASSEMBLER CONVENTIONS

**The standard 8080A assembler is available from 8080 manufacturers and on the major time-sharing networks; it is also part of most development systems. Cross assembler versions are available for most large computers and many minicomputers.**

## ASSEMBLER FIELD STRUCTURE

**The assembly language instructions have the standard field structure (see Figure 2-1 in Chapter 2). The required delimiters are:**

1) **A colon after a label, except for the pseudo-operations EQU, SET, and MACRO, which require a space.**
2) **A space after the operation code.**
3) **A comma between operands in the operand field. (remember this one!)**
4) **A semicolon before a comment.**

Typical 8080 assembly language instructions are:

```
START:  LDA     1000    ;GET LENGTH
        LXI     H,2300
        HLT
```

## LABELS

**The Assembler allows five characters in labels; the first character must be a letter, @, or ?. The other characters can be letters, numbers, @, or ?. We will only use letters and numbers.**

## PSEUDO-OPERATIONS

**The Assembler has the following pseudo-operations:**

```
DB      -       DEFINE BYTE
DS      -       DEFINE STORAGE
DW      -       DEFINE WORD
END
EQU     -       EQUATE
ORG     -       ORIGIN
SET
```

DB and DW are the DATA pseudo-operations used for placing data in ROM. DB is used for 8-bit data, DW for 16-bit data. The only unusual feature to remember is that DW stores the eight least significant bits of data in the first word and eight most significant bits in the second word. This is the standard 8080A/8085 procedure for storing addresses in memory, but is contrary to normal practice; you must be aware of the order when storing 16-bit data.

**Examples:**

```
ADDR:   DW      3165H
```

results in (ADDR) = 65, and (ADDR + 1) = 31 (hexadecimal).

```
TCONV:  DB      32
```

This pseudo-operation places the number 32 in the next byte of memory and assigns it the name TCONV.

```
ERROR:  DB      'ERROR'
```

This pseudo-operation places the ASCII characters E, R, R, O and R in the next five bytes of memory and assigns the name ERROR to the address of the first byte.

```
OPERS:  DW      FADD, FSUB, FMUL, FDIV
```

This pseudo-operation places the addresses FADD, FSUB, FMUL, and FDIV in the next eight bytes of memory and assigns the name OPERS to the address of the first byte.

DS is the RESERVE pseudo-operation used for assigning locations in RAM; it allocates 8-bit words (or bytes).

EQU is the EQUATE, or DEFINE pseudo-operation used to define names.

**EQU PSEUDO-OPERATION**

SET is similar to EQU, except that SET allows the name to be re-defined later. SET is unusual and confusing; it should be avoided unless it is required to patch programs together.

**ORG PSEUDO-OPERATION**

**ORG is the standard ORIGIN pseudo-operation. ORG 0 appearing at the beginning of a program may be omitted.**

**8080A/8085 programs usually have several origins;** they are used as follows:

1) To specify the RESET address (usually zero).
2) To specify Interrupt entry points (usually 0 to $3C_{16}$).
3) To specify the starting address of the main program.
4) To specify the starting addresses of subroutines.
5) To define RAM storage.
6) To define a RAM Stack.

**Example:**

```
RESET   EQU     0
        ORG     RESET
```

This sequence places the RESET instruction sequence in memory beginning at address 0.

```
INT1    EQU     38H
        ORG     INT1
```

The instruction sequence which follows is stored in memory beginning at location $38_{16}$.

END simply marks the end of the assembly language program.

## LABELS WITH PSEUDO-OPERATIONS

**The rules and recommendations for labels with 8080A/8085 pseudo-operations are as follows:**

1) EQU, SET and MACRO require labels, since the function of these pseudo-operations is to define the meaning of the label.
2) DB, DW and DS usually have labels (note that these labels are not followed by colons).
3) ORG, IF, ENDIF, ENDM, and END should not have labels since the meaning of the label is unclear.

## ADDRESSES

**NUMBERS AND CHARACTERS IN ADDRESS FIELD**

The Intel 8080A/8085 assemblers allow entries in the address field in any of the following forms:

1) Decimal (a final D is optional).
   Example: 1247 or 1247D.
2) Hexadecimal (must start with a digit and end with an H).
   Example: 124CH.
3) Octal (must end with O or Q but Q is far less confusing).
   Example: 1247Q or 1247O.
4) Binary (must end with B).
   Example: 1001001000111B.
5) ASCII (enclosed in single quotation marks).
   Example: 'HERE'.
6) As an offset from the Program Counter($).
   Example: $+237H.

**ASSEMBLER ARITHMETIC AND LOGICAL OPERATIONS**

All arithmetic and logical operations within an address field assume all arguments are 16-bit data; they produce 16-bit results. These operations are allowed as part of expressions in the address field:

1) + (16-bit addition)
2) - (16-bit subtraction)
3) * (multiplication, generating a 16-bit result).
4) / (integer quotient).
5) MOD (integer remainder).
6) NOT
7) AND
8) OR
9) XOR

   Standard Boolean operators

10) SHR (logical shift right).
11) SHL (logical shift left).

The shifts have the form:

```
SHR     OP1,OP2
SHL     OP1,OP2
```

where OP1 is the number to be shifted and OP2 is the number of shifts.

**ORDER OF ASSEMBLER OPERATIONS**

In address expressions with more than one of the operators listed above, the order of evaluation is defined below. Operators having the same precedence are evaluated left to right; expressions in parentheses are evaluated first. The order of precedence is:

Highest: 1) *, /, MOD, SHL, SHR
2) +, -
3) NOT
4) AND
Lowest: 5) OR, XOR

**We recommend that you avoid expressions within address fields wherever possible.** If you must compute an address, use the following guidelines:

1) Use parentheses to make expressions clearer. Do not depend on the order of operations.
2) Comment any unclear expressions.
3) Be sure that the evaluation of the expressions never produces a result which is too large for 16 bits.

## CONDITIONAL ASSEMBLY

**The 8080A/8085 assembler has a simple conditional assembly capability based on the pseudo-operations IF and ENDIF.** IF is followed by an expression, for example:

BASE-1000H or OPER1

If the expression is not zero, the assembler includes all the instructions up to the ENDIF pseudo-operation in the program; if the expression is zero, the assembler ignores all instructions between IF and ENDIF.

We will not use conditional assemblies, or refer to this capability again; it is sometimes handy for adding or eliminating debugging instructions, but that is all.

## MACROS

**The standard 8080A/8085 assembler has a macro capability which assigns names to instruction sequences.** Use the pseudo-operation MACRO to begin the definition and ENDM to end it. The macro may have parameters and may include any assembly language instructions except the definitions of other macros.

The macro capability is often a convenient programming shorthand, but we will not use it.

Note that instruction sequences defined by macros are generally quite short; they should not exceed ten or fifteen instructions. Longer sequences should be made into subroutines to conserve memory space.

Every MACRO pseudo-operation must have a label; the label is the name with which you identify the macro. For a discussion of this subject, see Chapter 2.

## BNPF FORMAT

**The 8080A/8085 assembler produces paper tapes in a format known as BNPF.** Binary data is represented by the ASCII codes for the letters B, N, P and F, as follows:

| | | |
|---|---|---|
| B | - | beginning of 8-bit word |
| N | - | binary 0 |
| P | - | binary 1 |
| F | - | ending of 8-bit word |

For example, the binary code 01101101 becomes:

BNPPNPPNPF

Thus ten bytes of paper tape are used to encode a single data byte. This format is used to create masks for ROMs and to program PROMs.

# Chapter 4
# SIMPLE PROGRAMS

**The only way to learn assembly language programming is to write assembly language programs. That is what we will do for the next six chapters, which contain examples of typical microprocessor tasks. Problems at the end of each chapter contain variations on the examples given in the text of the chapter. You should try to execute examples on an 8080A or 8085-based microcomputer to insure that you understand the material covered in the chapter.**

In this chapter we begin with some very simple programs.

## GENERAL FORMAT OF EXAMPLES

**Each program example contains the following parts:**

**EXAMPLE FORMAT**

1) A title which describes the problem.
2) A statement of purpose which describes the specific task which the program performs, plus the memory locations which it uses.
3) A sample problem showing input data and results.
4) A flowchart if the program logic is complex.
5) The source program, or assembly language listing of the program.
6) The object program, or hexadecimal machine language listing of the program.
7) Explanatory notes which discuss the instructions and methods used in the program.

**The problems at the end of the chapter are similar to the examples; problems should be programmed on an 8080A or 8085-based microcomputer system using the examples as guidelines.**

**The source programs in the examples have been constructed as follows:**

**GUIDELINES FOR EXAMPLES**

1) Standard Intel 8085 and 8080A assembler notation is used, as summarized in Chapter 3.
2) Data and address codes are selected for clarity rather than consistency. We use hexadecimal numbers for memory addresses, instruction codes, and BCD data, decimal for numeric constants, binary for logical masks, and ASCII for characters.
3) Frequently used instructions and programming techniques are emphasized.
4) Examples illustrate tasks which microprocessors actually perform in communications, instrumentation, computer, business equipment, industrial, and military applications.
5) Detailed comments are included.
6) Simple and clear structures are emphasized, but programs are as efficient as possible within this guideline. The notes often describe more efficient procedures.
7) Programs use consistent memory allocations. Each program starts in memory location 0 (the RESET location), and ends with the endless loop instruction:

```
HERE:  JMP  HERE
```

This endless loop instruction avoids difficulties associated with the 8080A HLT instruction. (i.e., systems which do not have interrupts may find it very difficult to clear HLT). You may replace this instruction with a RESTART or JUMP instruction which transfers control back to the monitor in some 8080-based microcomputers. Consult the user's manual for your microcomputer to determine the required memory allocations and terminating instruction.

## GUIDELINES FOR PROBLEMS

**When tackling the problems at the end of each chapter try to work within the following guidelines:**

**PROGRAMMING GUIDELINES**

1) Comment each program so that others can understand it. The comments can be brief and ungrammatical; they should explain the purpose of a section or instruction in the program. Comments should not describe the operation of instructions; that description is available in manuals. You do not have to comment each statement or explain the obvious. You may follow the format of the examples but provide less detail.
2) Emphasize clarity, simplicity, and good structure in programs. While programs should be reasonably efficient, do not worry about saving a single word of program memory or a few microseconds.
3) Make programs reasonably general. Do not confuse parameters (such as the number of elements in an array) with fixed constants (such as $\pi$ or ASCII C).
4) Never assume fixed initial values for parameters, i.e., use an instruction to load an initial value into a parameter. Do not place an initial value in the parameter as part of the object program.
5) Use assembler notation as shown in the examples and defined in Chapter 3.
6) Use hexadecimal notation for addresses. Use the clearest possible form for data.
7) If your microcomputer allows it, start all programs in memory location 0 and use memory locations starting with $40_{16}$ for data and temporary storage. Otherwise, establish equivalent addresses for your microcomputer and use them consistently. Again, consult the user's manual.
8) Use meaningful names for labels and variables, e.g., SUM or CHECK rather than X, Y, or Z.
9) Execute each program on your microcomputer. There is no other way of ensuring that your program is correct. We have provided sample data with each problem. Be sure that the program works for special cases.

**We now summarize some useful information that you should keep in mind when writing programs.**

**USING THE ACCUMULATOR**

Almost all processing instructions (e.g. ADD, SUBTRACT, AND, SHIFT, OR) use the Accumulator. In most cases you will load data into the Accumulator with either LDA or MOV A, M. You will store the result (from the Accumulator) in memory with either STA or MOV M, A.

**USING REGISTER M**

The preferred method of accessing memory is using implied addressing via Registers H and L, that is, using register code M. This code causes the 8080A or 8085 to perform a memory access using the address stored in Registers H and L. You can use LXI (Load Register Pair Immediate) or LHLD (Load H and L Direct) to place a starting value in Registers H and L. You can use INX (Increment Register Pair) or DCX (Decrement Register Pair) to increment or decrement (by 1) the address in Registers H and L.

The 8-bit arithmetic and logical operations all use the data in the Accumulator as one of their operands and place their result in the Accumulator.

Some of the 8-bit arithmetic and logical operations have special uses, for example:

**SPECIAL INSTRUCTIONS**

SUB A (or XRA A) clears the Accumulator.

ADD A shifts the Accumulator left logically. This instruction also multiplies the contents of the Accumulator by 2.

ANA A (or ORA A) clears the Carry flag while preserving the contents of the Accumulator.

A logical AND can mask off parts of a word. The required mask has '1' bits in the positions which you want to reserve and '0' bits in the positions which you want to clear.

## EXAMPLES

### Ones Complement

**Purpose:** Logically complement the contents of memory location $40_{16}$ and place the result in memory location $41_{16}$.

**Sample Problem:**

(40) = 6A
Result = (41) = 95

**Source Program:**

```
        LDA     40H     ;GET DATA
        CMA             ;COMPLEMENT DATA
        STA     41H     ;STORE RESULT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | Memory Contents (Hex) |
|---|---|---|
| 00 | LDA 40H | 3A |
| 01 | | 40 |
| 02 | | 00 |
| 03 | CMA | 2F |
| 04 | STA 41H | 32 |
| 05 | | 41 |
| 06 | | 00 |
| 07 | HERE: JMP HERE | C3 |
| 08 | | 07 |
| 09 | | 00 |

LDA, STA, and JMP all require 16-bit addresses; the eight least significant address bits are in the word immediately following the instruction code and the eight most significant bits in the next word (this order is unique to the 8080 and contrary to normal computer practice).

CMA is a one-word instruction which inverts each bit in the Accumulator. We often use CMA to transfer data to or from a peripheral device that uses negative logic (e.g., a display which is turned on by a zero, off by a one).

The address for the "endless loop" instruction (JMP HERE) is the address of the first word in that instruction. You will have to reset or stop the microcomputer in order to get it out of the loop.

## 8-Bit Addition

**Purpose:** Add the contents of memory locations 40 and 41 and place the result in memory location 42.

**Sample Problem:**

(40) = 38
(41) = 2B

Result = (42) = 63

**Source Program:**

```
        LXI     H,40H
        MOV     A,M     ;GET FIRST OPERAND
        INX     H
        ADD     M       ;ADD SECOND OPERAND
        INX     H
        MOV     M,A     ;STORE RESULT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | | LXI H,40H | 21 |
| 01 | | | 40 |
| 02 | | | 00 |
| 03 | | MOV A,M | 7E |
| 04 | | INX H | 23 |
| 05 | | ADD M | 86 |
| 06 | | INX H | 23 |
| 07 | | MOV M,A | 77 |
| 08 | HERE: | JMP HERE | C3 |
| 09 | | | 08 |
| 0A | | | 00 |

LXI H,40H loads the contents of the following two words of program memory into register pair H (Registers H and L). The first word goes into Register L, the second into Register H.

The register code 'M' means that data is obtained from or sent to the memory location addressed by Registers H and L. Thus MOV A,M moves data from the addressed location to the Accumulator; MOV M,A moves data from the Accumulator to the addressed location; ADD M adds the contents of the addressed location to the contents of the Accumulator. Remember that H and L contain a 16-bit address but that memory location contains 8 bits of data.

INX performs a 16-bit increment in one instruction cycle. The CPU doesn't use the 8-bit arithmetic unit for the increment; it uses the incrementer that it normally uses to increment the 16-bit Program Counter.

MOV A,M and MOV M,A are preferable to LDA and STA whenever you use the same memory location repeatedly or use adjacent locations, because MOV requires less program memory and time than LDA and STA. Note, however, that you must load Registers H and L before you can use Register M.

## Shift Left One Bit

**Purpose:** Shift the contents of memory location 40 left one bit and place the result in memory location 41. Clear the empty bit position.

**Sample Problem:**

(40) = 6F

Result = (41) = DE

**Source Program:**

```
        LDA     40H     ;GET DATA
        ADD     A       ;SHIFT DATA LEFT LOGICALLY
        STA     41H     ;STORE RESULT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | Memory Contents (Hex) |
|---|---|---|
| 00 | LDA 40H | 3A |
| 01 | | 40 |
| 02 | | 00 |
| 03 | ADD A | 87 |
| 04 | STA 41H | 32 |
| 05 | | 41 |
| 06 | | 00 |
| 07 | HERE: JMP HERE | C3 |
| 08 | | 07 |
| 09 | | 00 |

ADD A simply adds the contents of the Accumulator to itself. The result is, of course, twice the original data, which is the same result that a logical left shift would produce. The least significant bit of the result is zero since 0+0 = 1+1 = 0; 1+1 also produces a Carry to the next bit.

## Mask Out Least Significant Four Bits

**Purpose:** Place the least significant four bits of the contents of memory location 40 in memory location 41. Clear the most significant four bits of memory location 41.

**Sample Problem:**

(40) = B8

Result = (40) = 08

**Source Program:**

```
        LDA     40H             ;GET DATA
        ANI     00001111B       ;MASK OUT LEAST SIGNIFICANT BITS
        STA     41H             ;STORE RESULT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | LDA | 40H | 3A |
| 01 | | | 40 |
| 02 | | | 00 |
| 03 | ANI | 00001111B | E6 |
| 04 | | | 0F |
| 05 | STA | 41H | 32 |
| 06 | | | 41 |
| 07 | | | 00 |
| 08 | HERE: JMP | HERE | C3 |
| 09 | | | 08 |
| 0A | | | 00 |

ANI logically ANDs the contents of the Accumulator with the contents of the word of program memory immediately following the instruction. AND may be used to clear bits that are not in use. The four least significant bits could be an input from a switch or an output to a numeric display.

## Clear A Memory Location

**Purpose**: Clear memory location 40.

**Source Program:**

```
        SUB     A
        STA     40H     ;CLEAR LOCATION 40
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | Memory Contents (Hex) |
|---|---|---|
| 00 | SUB A | 97 |
| 01 | STA 40H | 32 |
| 02 | | 40 |
| 03 | | 00 |
| 04 | HERE: JMP HERE | C3 |
| 05 | | 04 |
| 06 | | 00 |

SUB A subtracts the number in the Accumulator from itself. The result is to clear the Accumulator. This instruction, XRA A, or MVI A,0 can all clear the Accumulator. MVI A,0 takes more time and memory but doesn't affect the status flags.

## Word Disassembly

**Purpose**: Divide the contents of memory location 40 into two 4-bit sections and store them in memory locations 41 and 42. Place the four most significant bits of memory location 40 in the four least significant bit positions of memory location 41; place the four least significant bits of memory location 40 in the four least significant bit positions of memory location 42. Clear the four most significant bit positions of memory locations 41 and 42.

**Sample Problem:**

(40) = 3F

Result = (41) = 03
(42) = 0F

**Source Program:**

```
        LXI     H,40H
        MOV     A,M             ;GET DATA
        MOV     B,A
        RRC
        RRC                     ;SHIFT DATA RIGHT 4 TIMES
        RRC
        RRC
        ANI     00001111B       ;MASK OFF MSBs
        INX     H
        MOV     M,A             ;STORE MSBs
        MOV     A,B             ;RESTORE ORIGINAL DATA
        ANI     00001111B       ;MASK OFF LSBs
        INX     H
        MOV     M,A             ;STORE LSBs
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MOV | A,M | 7E |
| 04 | | MOV | B,A | 47 |
| 05 | | RRC | | 0F |
| 06 | | RRC | | 0F |
| 07 | | RRC | | 0F |
| 08 | | RRC | | 0F |
| 09 | | ANI | 00001111B | E6 |
| 0A | | | | 0F |
| 0B | | INX | H | 23 |
| 0C | | MOV | M,A | 77 |
| 0D | | MOV | A,B | 78 |
| 0E | | ANI | 00001111B | E6 |
| 0F | | | | 0F |
| 10 | | INX | H | 23 |
| 11 | | MOV | M,A | 77 |
| 12 | HERE: | JMP | HERE | C3 |
| 13 | | | | 12 |
| 14 | | | | 00 |

Instructions using Register M occupy only one word of program memory. However, the Address register (H and L) must be loaded before M can be used. Thus implied memory addressing saves time and memory, as compared to direct memory addressing, only when the program repeatedly uses the same address or consecutive addresses.

RRC shifts the Accumulator right one bit circularly with the least significant bit going to the most significant bit position and to the Carry. Shifting the Accumulator right four times requires four RRCs.

Many 8080A/8085 instructions affect a pair of 8-bit registers. The pairs are H (H and L), D (D and E), B (B and C), and SP (the Stack Pointer). Registers B, D and H are the most significant 8 bits of the pairs; Registers C, E, and L are the least significant 8 bits. The common instructions that use register pairs are LXI (Load Register Pair Immediate), DCX (16-Bit Decrement), INX (Increment A Register Pair), and DAD (Add A Register Pair To H and L).

## Find Larger Of Two Numbers

**Purpose:** Place the larger of the contents of memory locations 40 and 41 in memory location 42. Assume that the contents of memory locations 40 and 41 are unsigned binary numbers.

**Sample Problems:**

a. (40) = 3F
(41) = 2B

Result = (42) = 3F

b. (40) = 75
(41) = A8

Result = (42) = A8

**Source Program:**

```
        LXI     H,40H
        MOV     A,M         ;GET FIRST OPERAND
        INX     H
        CMP     M           ;IS SECOND OPERAND LARGER?
        JNC     DONE
        MOV     A,M         ;YES, GET SECOND OPERAND INSTEAD
DONE:   INX     H
        MOV     M,A         ;STORE LARGER OPERAND
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | | LXI H,40H | 21 |
| 01 | | | 40 |
| 02 | | | 00 |
| 03 | | MOV A,M | 7E |
| 04 | | INX H | 23 |
| 05 | | CMP M | BE |
| 06 | | JNC DONE | D2 |
| 07 | | | 0A |
| 08 | | | 00 |
| 09 | | MOV A,M | 7E |
| 0A | DONE: | INX H | 23 |
| 0B | | MOV M,A | 77 |
| 0C | HERE: | JMP HERE | C3 |
| 0D | | | 0C |
| 0E | | | 00 |

CMP M sets the flags as if the contents of the memory locatation addressed by H and L had been subtracted from the contents of the Accumulator. However, the contents of the Accumulator are left unchanged for later comparisons or other processing.

If A is the contents of the Accumulator and X is the second operand for a CMP or CPI instruction, then flags are set as follows:

1) Zero = 1 if A = X
Zero = 0 if A ≠ X

2) Carry = 1 if A < X
Carry = 0 if A ≥ X
(A, X unsigned binary numbers)

CMP sets the Carry to 1 if a Borrow would be necessary to actually perform the subtraction, i.e., if the number being subtracted from the contents of the Accumulator is greater than those contents. Thus the sequence CMP, JNC causes a Jump if the contents of the Accumulator are greater than or equal to the other number.

JNC DONE causes a Jump to memory location DONE if the Carry flag = 0. Otherwise (if Carry = 1), the computer continues with the next sequential memory location after the JNC instruction.

DONE and HERE are just names which you assign to locations in program memory so that they are easier to remember. Remember that these names are labels and must be followed by a colon on the line where they are defined. In this program, DONE is memory location 000A and HERE is memory location 000C. The 8080A/8085 assemblers allow five characters in labels — the first must be a letter while the others may be letters or numbers (some special characters are allowed but we will not use them).

## 16-Bit Addition

**Purpose:** Add the 16-bit number in memory locations 40 and 41 to the 16-bit number in memory locations 42 and 43. The most significant eight bits of the two numbers to be added are in memory locations 41 and 43. Store the result in memory locations 44 and 45 with the most significant byte in memory location 45.

**Sample Problem:**

(40) = 2A
(41) = 67
(42) = F8
(43) = 14

Result = 672A+14F8 = 7C22
(44) = 22
(45) = 7C

**Source Program:**

```
        LHLD    40H     ;GET FIRST 16-BIT NUMBER
        XCHG
        LHLD    42H     ;GET SECOND 16-BIT NUMBER
        DAD     D       ;16-BIT ADDITION
        SHLD    44H     ;STORE 16-BIT RESULT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | LHLD | 40H | 2A |
| 01 | | | 40 |
| 02 | | | 00 |
| 03 | XCHG | | EB |
| 04 | LHLD | 42H | 2A |
| 05 | | | 42 |
| 06 | | | 00 |
| 07 | DAD | D | 19 |
| 08 | SHLD | 44H | 22 |
| 09 | | | 44 |
| 0A | | | 00 |
| 0B | HERE: JMP | HERE | C3 |
| 0C | | | 0B |
| 0D | | | 00 |

LHLD loads Registers H and L from two memory locations, the one specified in the instruction and the next consecutive one.The contents of the first addressed location goes to Register L. The contents of the next location goes to Register H. Thus LHLD 40 means (L) = (40), (H) = (41). The actual transfer proceeds one byte at a time and takes 16 clock cycles. The advantage of the 16-bit Load instructions over two 8-bit Load instructions is that the CPU only has to fetch one instruction from memory. Note the difference between LXI, which loads a fixed 16-bit value from ROM into a register pair, and LHLD, which loads the contents of two RAM locations into H and L.

XCHG exchanges the contents of Registers D and E with H and L. No numbers are changed or destroyed. XCHG can replace a whole series of MOV instructions.

DAD adds the 16-bit number in Registers D and E to the 16-bit number in Registers H and L. The result is placed in Registers H and L. DAD actually adds one byte at a time; it executes in ten clock cycles.

SHLD stores the contents of Registers H and L in two memory locations, the one specified in the instruction and the next consecutive one. The contents of L goes in the specified location and the contents of H goes in the next location. Thus SHLD 44 means (44) = (L), (45) = (H). As with LHLD, the actual transfer proceeds one byte at a time; it executes in 16 clock cycles.

Although the 8080 is an 8-bit processor, it has instructions which handle 16-bit numbers. These instructions are primarily intended for handling addresses, but you can also use them for 16-bit data. The most common ones and their uses are:

1) DAD — 16-Bit Add
   Used to access tables and to add 16-bit data units.
2) DCX — 16-Bit Decrement
   Used to subtract 1 from the contents of an Address register.
3) INX — 16-Bit Increment
   Used to add 1 to the contents of an Address register.
4) LHLD — 16-Bit Load Direct
   Used to place variable addresses in the main Address register (H and L).
5) LXI — 16-Bit Load Immediate
   Used to initialize an Address register with a fixed value, i.e., the starting address of an array or table.
6) SHLD — 16-Bit Store Direct
   Used to store addresses in memory from the main Address register (H and L).

## Table Of Squares

**Purpose:** Calculate the square of the contents of memory location 40, using a table, and place the result in memory location 41. Assume memory location 40 contains a number between 0 and 7 inclusive, i.e., $0 \leq (40) \leq 7$.

The table occupies memory locations 60 to 67.

| Memory Address (Hex) | Entry (Hex) | (Decimal) |
|---|---|---|
| 60 | 00 | 0 $(0^2)$ |
| 61 | 01 | 1 $(1^2)$ |
| 62 | 04 | 4 $(2^2)$ |
| 63 | 09 | 9 $(3^2)$ |
| 64 | 10 | 16 $(4^2)$ |
| 65 | 19 | 25 $(5^2)$ |
| 66 | 24 | 36 $(6^2)$ |
| 67 | 31 | 49 $(7^2)$ |

**Sample Problems:**

a. (40) = 03
Result = (41) = 09

b. (40) = 06
Result = (41) = 24

**Source Program:**

```
        LDA     40H     ;GET DATA
        MOV     L,A     ;MAKE DATA INTO 16-BIT INDEX
        MVI     H,0
        LXI     D,60H   ;GET STARTING ADDRESS OF TABLE OF SQUARES
        DAD     D       ;INDEX TABLE WITH DATA
        MOV     A,M     ;GET SQUARE OF DATA
        STA     41H
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | LDA | 40H | 3A |
| 01 | | | 40 |
| 02 | | | 00 |
| 03 | MOV | L,A | 6F |
| 04 | MVI | H,0 | 26 |
| 05 | | | 00 |
| 06 | LXI | D,60H | 11 |
| 07 | | | 60 |
| 08 | | | 00 |
| 09 | DAD | D | 19 |
| 0A | MOV | A,M | 7E |
| 0B | STA | 41H | 32 |
| 0C | | | 41 |
| 0D | | | 00 |
| 0E | HERE: JMP | HERE | C3 |
| 0F | | | 0E |
| 10 | | | 00 |

Note that you must also enter the table of squares into memory (the assembler pseudo-operation DB will handle this). The table of squares represents constant data, not parameters that may change; that is why you can initialize the table of squares using the DB pseudo-operation, rather than by executing instructions to load values into the table.

MOV L,A moves the data in the Accumulator to Register L. The data is the eight least significant bits of the index.

MVI H,0 clears Register H so that it does not interfere with the 16-bit addition of starting address and index. Never assume that a register contains zero at the start of a program.

LXI D,60H loads the starting address of the table into Registers D and E. We use D and E for the starting address since the DAD instruction does not change D and E. Thus the starting address of the table will still be in D and E in the event that we want another element from the table.

DAD D adds the starting address and the index; the result in H and L is thus the address of the correct entry. MOV A,M then moves that entry to the Accumulator.

Arithmetic which a microprocessor cannot do directly in a few instructions is often best performed with lookup tables. Lookup tables simply contain all the possible answers to the problem; they are organized so that the answer to a particular problem can be found easily. The arithmetic problem now becomes an accessing problem — how do we get the correct answer from the table? We must know two things — the position of the answer in the table (called the index) and the base or starting address of the table. The address of the answer is then just the base address plus the index.

The base address, of course, is a fixed number for a particular table. How can we determine the index? In simple cases, where a single piece of data is involved, we can organize the table so that the data is the index. In the table of squares, the 0th entry in the table contains zero squared, the first entry 1 squared, etc. In more complex cases where the spread of input values is very large or there are several data items involved (e.g., roots of a quadratic or number of permutations), we must use more complicated methods to determine indexes.

The basic tradeoff in using a table is time vs. memory. Tables are faster, since no computations are required, and simpler, since no mathematical methods must be devised and tested. However, tables can occupy a large amount of memory if the range of the input data is large. We can often reduce the size of a table by limiting the accuracy of the results, scaling the input data, or organizing the table cleverly. Tables are often used to compute transcendental and trigonometric functions, linearize inputs, convert codes, and perform other mathematical tasks.

## 16-Bit Ones Complement

**Purpose:** Place the ones complement of the 16-bit number in memory locations 40 and 41 into memory locations 42 and 43. The most significant bytes are in locations 41 and 43.

**Sample Problem:**

(40) = 67
(41) = E2

Result = (42) = 98
(43) = 1D

The ones complement inverts each bit of the original number; the sum of the original number and its ones complement will always be all 1 bits.

**Source Program:**

```
        LHLD    40H     ;GET DATA
        MOV     A,L     ;COMPLEMENT 8 LSBs
        CMA
        MOV     L,A
        MOV     A,H     ;COMPLEMENT 8 MSBs
        CMA
        MOV     H,A
        SHLD    42H     ;STORE ONES COMPLEMENT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | LHLD | 40H | 2A |
| 01 | | | 40 |
| 02 | | | 00 |
| 03 | MOV | A,L | 7D |
| 04 | CMA | | 2F |
| 05 | MOV | L,A | 6F |
| 06 | MOV | A,H | 7C |
| 07 | CMA | | 2F |
| 08 | MOV | H,A | 67 |
| 09 | SHLD | 42H | 22 |
| 0A | | | 42 |
| 0B | | | 00 |
| 0C | HERE: JMP | HERE | C3 |
| 0D | | | 0C |
| 0E | | | 00 |

## PROBLEMS

### 1) Twos Complement

**Purpose:** Place the twos complement of the contents of memory location 40 into memory location 41. The twos complement is the ones complement plus one.

**Sample Problem:**

(40) = 3E

Result = (41) = D2

The sum of the original number and its twos complement is zero (try the sample case).

### 2) 8-Bit Subtraction

**Purpose:** Subtract the contents of memory location 41 from the contents of memory location 40. Place the result in memory location 42.

**Sample Problem:**

(40) = 77
(41) = 39

Result = (42) = 3E

### 3) Shift Left Two Bits

**Purpose:** Shift the contents of memory location 40 left two bits and place the result in memory location 41. Clear the two least significant bit positions.

**Sample Problem:**

(40) = 5D

Result = (41) = 74

### 4) Mask Out Most Significant Four Bits

**Purpose:** Place the four most significant bits of the contents of memory location 40 into memory location 41. Clear the four least significant bits of memory location 41.

**Sample Problem:**

(40) = C4

Result = (41) = C0

### 5) Set A Memory Location To All Ones

**Purpose:** Memory location 40 is set to all ones (FF hex).

### 6) Word Assembly

**Purpose:** Combine the four least significant bits of memory locations 40 and 41 into a word and store them in memory location 42. Place the four least significant bits of memory location 40 in the four most significant bit positions of memory location 42; place the four least significant bits of memory location 41 in the four least significant bit positions of memory location 42.

**Sample Problem:**

(40) = 6A
(41) = B3

Result = (42) = A3

## 7) Find Smaller Of Two Numbers

**Purpose:** Place the smaller of the contents of memory locations 40 and 41 in memory location 42. Assume that 40 and 41 contain unsigned binary numbers.

**Sample Problems:**

a.
(40) = 3F
(41) = 2B

Result = (42) = 2B

b.
(40) = 75
(41) = A8

Result = (42) = 75

## 8) 24-Bit Addition

**Purpose:** Add the 24-bit number in memory locations 40, 41, and 42 to the 24-bit number in memory locations 43, 44, and 45. The most significant eight bits are in memory locations 42 and 45, the least significant eight bits in memory locations 40 and 43. Store the result in memory locations 46, 47, and 48 with the most significant bits in 48 and the least significant bits in 46.

**Sample Problem:**

(40) = 2A
(41) = 67
(42) = 35

(43) = F8
(44) = A4
(45) = 51

Result = (46) = 22
(47) = 0C
(48) = 87

i.e., 35672A
\+ 51A4F8
870C22

## 9) Sum Of Squares

**Purpose:** Calculate the squares of the contents of memory locations 40 and 41 and add them together. Place the result in memory location 42. Assume that memory locations 40 and 41 both contain a number between 0 and 7 inclusive, i.e., $0 \leq (40) \leq 7$ and $0 \leq (41) \leq 7$. Use the table of squares from the example entitled "Table Of Squares".

**Sample Problem:**

(40) = 03
(41) = 06

Result = (42) = 2D
i.e., $3^2+6^2 = 9+36$ (decimal)
$= 45 = 2D$ (hex)

## 10) 16-Bit Twos Complement

**Purpose**: Place the twos complement of the 16-bit number in memory locations 40 and 41 (most significant bits in 41) in memory locations 42 and 43 (most significant bits in 43).

**Sample Problem:**

(40) = 00
(41) = 58

Result = (42) = 00
(43) = A8

# Chapter 5
# SIMPLE PROGRAM LOOPS

The program loop is the basic structure which forces the CPU to repeat a sequence of instructions. Loops have four sections:

1) The initialization section which establishes the starting values of counters, Address Registers (pointers), and other variables.
2) The processing section where the actual data manipulation occurs. This is the section which does the work.
3) The loop control section which updates counters and pointers for the next iteration.
4) The concluding section which analyzes and stores the results.

Note that the computer performs Sections 1 and 4 only once, while it may perform Sections 2 and 3 many times. Thus, the execution time of the loop will be mainly dependent on the execution time of Sections 2 and 3. You will want Sections 2 and 3 to execute as quickly as possible; do not worry about the execution time of Sections 1 and 4. A typica program loop can be flowcharted as shown in Figure 5-1, or the positions of the processing and loop control sections may be reversed as shown in Figure 5-2. The processing section in Figure 5-1 is always executed at least once, while the processing section in Figure 5-2 may not be executed at all. Figure 5-1 seems more natural, but Figure 5-2 is often more efficient and avoids the problem of what to do when there is no data (a bugaboo for computers and the frequent cause of silly situations like the computer dunning someone for a bill of $0.00).

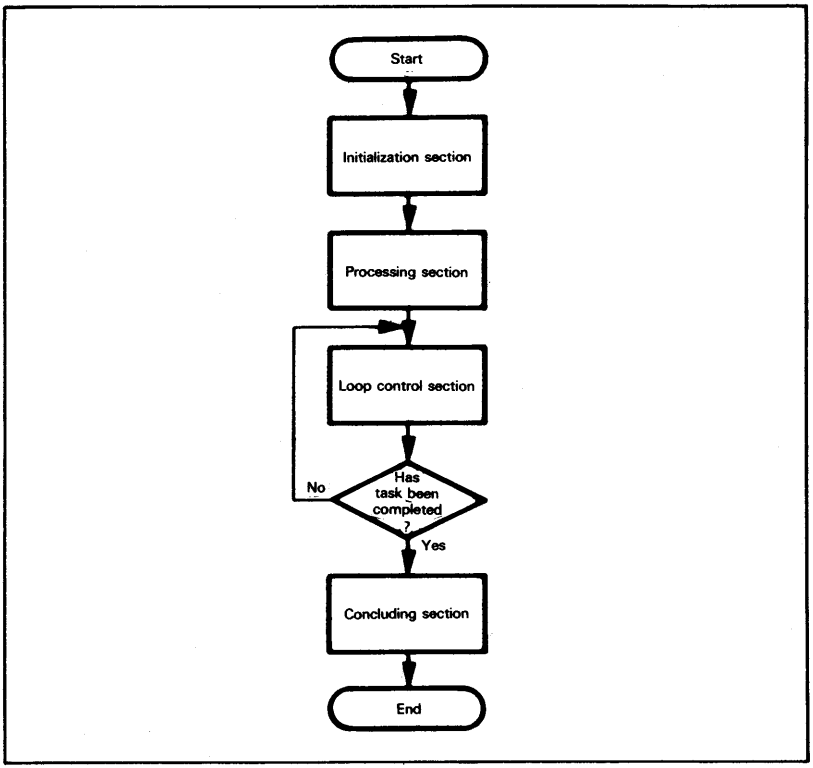



Figure 5-1. Flowchart Of A Program Loop

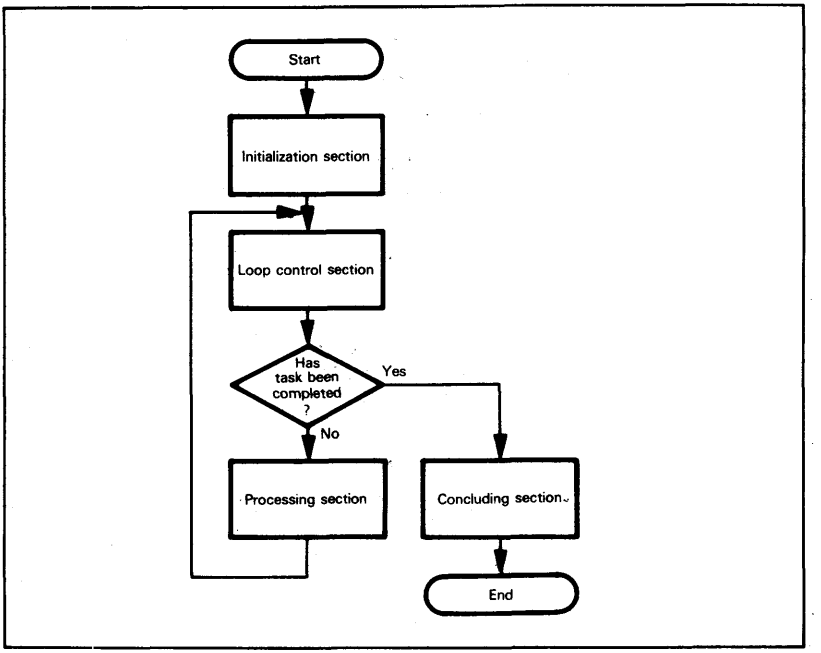



Figure 5-2. A Program Loop Which Allows Zero Iterations

The loop structure can process entire blocks of data. To accomplish this, the program must increment an Address register (usually register pair H) after each iteration so that the Address register points to the next element in the data block. The next iteration will then perform the same operations on the data in the next memory location. The computer can handle blocks of any length with the same set of instructions. Register M is the key to processing blocks of data with the 8080, since it allows you to vary the actual memory address by changing the contents of register pair H.

## EXAMPLES

### Sum Of Data

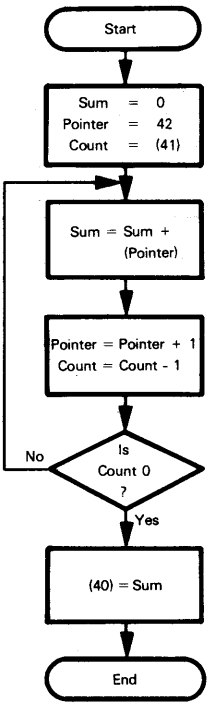
**8-BIT SUMMATION**

**Purpose:** Calculate the sum of a series of numbers. The length of the series is in memory location 41 and the series itself begins in memory location 42. Store the sum in memory location 40. Assume the sum to be an 8-bit number so you can ignore carries.

**Sample Problem:**

(41) = 03
(42) = 28
(43) = 55
(44) = 26
Result = (42) + (43) + (44) = (40)
28 + 55 + 26 = A3
There are 3 entries in the sum since (41) = 03.

**Flowchart:**



**Source Program:**

```
        LXI     H,41H
        MOV     B,M     ;COUNT = LENGTH OF SERIES OF NUMBER
        SUB     A       ;SUM = 0
SUMD:   INX     H
        ADD     M       ;SUM = SUM + DATA
        DCR     B
        JNZ     SUMD
        STA     40H     ;STORE SUM
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,41H | 21 |
| 01 | | | | 41 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | SUB | A | 97 |
| 05 | SUMD: | INX | H | 23 |
| 06 | | ADD | M | 86 |
| 07 | | DCR | B | 05 |
| 08 | | JNZ | SUMD | C2 |
| 09 | | | | 05 |
| 0A | | | | 00 |
| 0B | | STA | 40H | 32 |
| 0C | | | | 40 |
| 0D | | | | 00 |
| 0E | HERE: | JMP | HERE | C3 |
| 0F | | | | 0E |
| 10 | | | | 00 |

The first three instructions form the initialization section of the program. They set the pointer, counter, and sum to their starting values. Note that MOV can transfer data between memory and any of the General Purpose registers, while LDA and STA can only transfer data between memory and the Accumulator.

The processing section of the program is the single instruction ADD M which adds the contents of the memory location being addressed by Registers H and L to the contents of the Accumulator, and stores the result in the Accumulator. This instruction does the real work of the program.

The loop control section of the program consists of the instructions INX H and DCR B. INX H updates the pointer so that the next iteration adds the next number to the sum. DCR B decrements the counter which keeps track of how many iterations are left.

The instruction JNZ SUMD causes a Jump to location SUMD if the Zero flag is zero. If the Zero flag is one, the CPU executes the next instruction in sequence (i.e., STA 40H). Since DCR B was the last instruction before JNZ to affect the Zero flag, JNZ SUMD causes a Jump to SUMD if DCR B did not produce a zero result, i.e.,

(PC) = SUMD if (B) ≠0
(PC) = (PC) + 3 if (B) =0

(PC is incremented by 3 because of the three-word JNZ instruction.) A single instruction which combined the Decrement and the Jump would be a useful addition to the 8080 instruction set.

Most computer loops count down rather than up so that the Zero flag can be used as an exit condition. Remember that the Zero flag is 1 if the result was zero and 0 if the result was not zero. Try rewriting the program so that it counts up rather than down; which method is more efficient?

The order of instructions is often very important. For example, DCR B must come right before JNZ SUMD, since otherwise the Zero result set by DCR B could be changed by another instruction. In addition, INX H must come before ADD M or else the first number added to the sum will be (41) instead of (42).

## 16-Bit Sum Of Data

**Purpose:** Calculate the sum of a series of numbers. The length of the series is in memory location 42, and the series itself begins in memory location 43. Store the sum in memory locations 40 and 41 (eight most significant bits in 41).

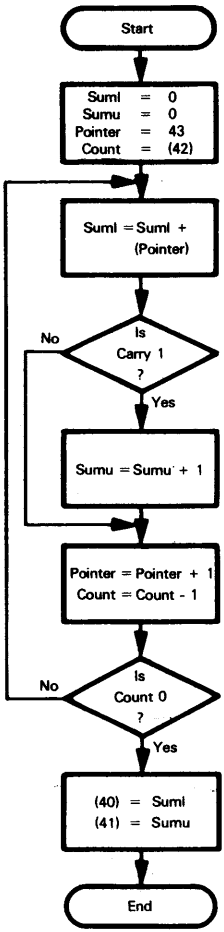
**Sample Problem:**

(42) = 03
(43) = C8
(44) = FA
(45) = 96

Result = C8 + FA + 96 = 0258 (hex)

(40) = 58
(41) = 02

**Flowchart:**



**Source Program:**

```
        LXI     H,42H
        MOV     B,M     ;COUNT = LENGTH OF SERIES
        SUB     A       ;LSBs OF SUM = 0
        MOV     C,A     ;MSBs OF SUM = 0
DSUMD:  INX     H
        ADD     M       ;SUM = SUM + DATA
        JNC     CHCNT
        INR     C       ;ADD CARRY TO MSBs OF SUM
CHCNT:  DCR     B
        JNZ     DSUMD
        LXI     H,40H   ;STORE LSBs OF SUM
        MOV     M,A
        INX     H       ;STORE MSBs OF SUM
        MOV     M,C
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,42H | 21 |
| 01 | | | | 42 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | SUB | A | 97 |
| 05 | | MOV | C,A | 4F |
| 06 | DSUMD: | INX | H | 23 |
| 07 | | ADD | M | 86 |
| 08 | | JNC | CHCNT | D2 |
| 09 | | | | 0C |
| 0A | | | | 00 |
| 0B | | INR | C | 0C |
| 0C | CHCNT: | DCR | B | 05 |
| 0D | | JNZ | CHCNT | C2 |
| 0E | | | | OC |
| 0F | | | | 00 |
| 10 | | LXI | H,40H | 21 |
| 11 | | | | 40 |
| 12 | | | | 00 |
| 13 | | MOV | M,A | 77 |
| 14 | | INX | H | 23 |
| 15 | | MOV | M,C | 71 |
| 16 | HERE: | JMP | HERE | C3 |
| 17 | | | | 16 |
| 18 | | | | 00 |

The structure of this program is the same as the structure of the last one. The most significant bits of the sum now must be initialized and stored. The processing section consists of three instructions (ADD M, JNC CHCNT, INR C), including a conditional Jump.

JNC CHCNT causes a Jump to memory location CHCNT if the Carry = 0. Thus, if there is no Carry from the 8-bit addition, the program jumps around the statement that increments the most significant bits of the sum.

INR C adds 1 to the contents of Register C. INR is an 8-bit increment; INX is a 16-bit increment.

## Number Of Negative Elements

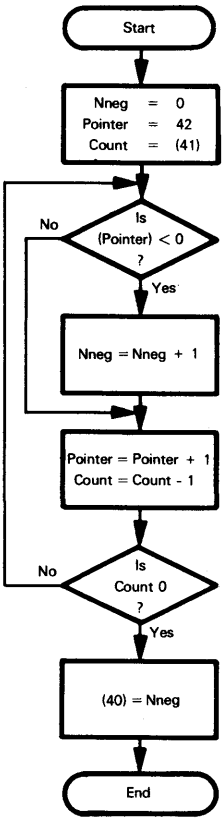
**Purpose:** Determine the number of negative elements (most significant bit 1) in a block of data. The length of the block is in memory location 41 and the block itself starts in memory location 42. Store the number of negative elements in memory location 40.

**Sample Problem:**

(41) = 06
(42) = 68
(43) = F2
(44) = 87
(45) = 30
(46) = 59
(47) = 2A

Result = (40) =02 since 43 and 44 contain numbers with a MSB of 1.

**Flowchart:**



**Source Program:**

```
        LXI     H,41H
        MOV     B,M             ;COUNT = NUMBER OF ELEMENTS
        MVI     A,01111111B     ;LARGEST POSITIVE NUMBER FOR
                                ;COMPARISON
        MVI     C,0             ;NUMBER OF NEGATIVES =0
SRNEG:  INX     H
        CMP     M               ;IS MSB OF DATA 1?
        JNC     CHCNT           ;NO IF NO BORROW NEEDED
        INR     C               ;YES, ADD 1 TO NUMBER OF NEGATIVES
CHCNT:  DCR     B
        JNZ     SRNEG
        MOV     A,C
        STA     40H             ;STORE NUMBER OF NEGATIVES
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,41H | 21 |
| 01 | | | | 41 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | MVI | A,01111111B | 3E |
| 05 | | | | 7F |
| 06 | | MVI | C,0 | 0E |
| 07 | | | | 00 |
| 08 | SRNEG: | INX | H | 23 |
| 09 | | CMP | M | BE |
| 0A | | JNC | CHCNT | D2 |
| 0B | | | | 0E |
| 0C | | | | 00 |
| 0D | | INR | C | 0C |
| 0E | CHCNT: | DCR | B | 05 |
| 0F | | JNZ | SRNEG | C2 |
| 10 | | | | 08 |
| 11 | | | | 00 |
| 12 | | MOV | A,C | 79 |
| 13 | | STA | 40H | 32 |
| 14 | | | | 40 |
| 15 | | | | 00 |
| 16 | HERE: | JMP | HERE | C3 |
| 17 | | | | 16 |
| 18 | | | | 00 |

Negative numbers all have a most significant bit of 1. Thus negative numbers are actually larger than positive numbers when twos complement notation is used.

CMP M sets the flags as if the contents of the memory location addressed by Registers H and L had been subtracted from the contents of the Accumulator. Neither the Accumulator nor the memory location is changed. Thus the value in the Accumulator stays the same for the next iteration.

CMP M sets flags as follows:

Zero = 1 if (A) = (M)
= 0 if (A) ≠ (M)

Carry = 1 if (A) < (M)
= 0 if (A) ≥ (M), i.e., the subtraction does not require a Borrow.

Since A contains 7F (hex) in the program, the flags will be:

Zero = 1 if (M) = 7F
= 0 if (M) ≠ 7F

Carry = 1 if (M) > 7F
= 0 if (M) ≤ 7F

So if (M) has an MSB of 1, the Carry will be 1; if (M) has an MSB of 0, the Carry will be 0.

The CMP instruction is necessary in order to set the flags. Placing the data in the Accumulator (MOV A, M) doesn't affect the flags; we would have to follow it with ORA A or ANA A, which affect the flags but don't change the value in the Accumulator. CMP is convenient because it affects the flags without changing any registers and can thus be used to d rectly examine the contents of a memory location. The CMP instruction also affects the Sign flag. If you replace the JNC CHCNT instruction with a JM CHCNT instruction, can you restructure the initialization section of the source program so that it will perform the specified task?

## Find Maximum

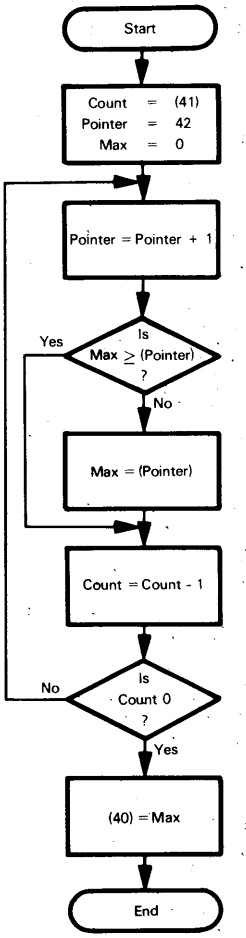
**Purpose:** Find the largest element in a block of data. The length of the block is in memory location 41 and the block itself begins in memory location 42. Store the maximum in memory location 40. Assume that the numbers in the block are all 8-bit unsigned binary numbers.

**Sample Problem:**

(41) = 05
(42) = 67
(43) = 79
(44) = 15
(45) = E3
(46) = 72

Result = (40) = E3 since this is the largest of the five unsigned numbers.

**Flowchart:**




**Source Program:**

```
        LXI     H,41H     ;POINT TO COUNT
        MOV     B,M       ;COUNT = NUMBER OF ELEMENTS
        SUB     A         ;MAXIMUM = MINIMUM POSSIBLE VALUE (ZERO)
NEXTE:  INX     H
        CMP     M         ;IS NEXT ELEMENT > MAXIMUM?
        JNC     DECNT
        MOV     A,M       ;YES, REPLACE MAXIMUM
DECNT:  DCR     B
        JNZ     NEXTE
        STA     40H       ;SAVE MAXIMUM
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | SUB | A | 97 |
| 05 | NEXTE: | INX | H | 23 |
| 06 | | CMP | M | BE |
| 07 | | JNC | DECNT | D2 |
| 08 | | | | 0B |
| 09 | | | | 00 |
| 0A | | MOV | A,M | 7E |
| 0B | DECNT: | DCR | B | 05 |
| 0C | | JNZ | NEXTE | C2 |
| 0D | | | | 05 |
| 0E | | | | 00 |
| 0F | | STA | 40H | 32 |
| 10 | | | | 40 |
| 11 | | | | 00 |
| 12 | HERE: | JMP | HERE | C3 |
| 13 | | | | 12 |
| 14 | | | | 00 |

The first three bytes of this program form the initialization section. The SUB A clears the Accumulator.

This program takes advantage of the fact that zero is the smallest 8-bit unsigned binary number. When you set the register that will contain the maximum value, in this case the Accumulator, to the minimum possible value before you enter the loop, then the program will set the Accumulator to a larger value unless all the elements in the list are zeros.

The program works properly if there is only one element but not if there are none at all. Why? How could you solve this problem?

The instruction CMP M sets the Carry flag as follows:

$$\begin{aligned} \text{Carry} &= 1 \text{ if ELEMENT} > \text{MAX} \\ &= 0 \text{ if ELEMENT} \leq \text{MAX} \end{aligned}$$

If Carry = 0, the program proceeds to DECNT and does not change the maximum. If Carry = 1, the program replaces the old maximum with the current element.

The program does not work if the numbers are signed because negative numbers will appear to be larger than positive numbers. The problem is somewhat tricky because of overflow; you can solve it by remembering that overflow can only occur in subtraction when the numbers have the opposite sign. Why? If the numbers do have the opposite sign, the positive number is clearly larger. The solution would be simpler if the 8080 had an Overflow bit.

## Justify A Binary Fraction

**Purpose:** Shift the contents of memory location 40 left until the most significant bit of the number is 1. Store the result in memory location 41 and the number of left shifts required in memory location 42. If the contents of memory location 40 are zero, clear both 41 and 42.

Note that the process is just like converting a number to a scientific notation, e.g., $0.0057 = 5.7 \times 10^{-3}$.

**Sample Problems:**

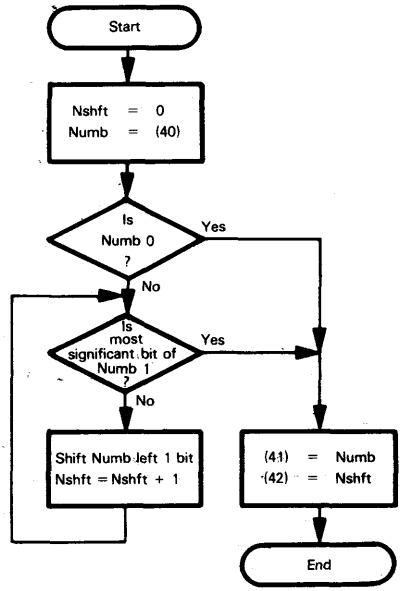
a. (40) = 22
Result = (41) = 88
(42) = 02

b. (40) = 01
Result = (41) = 80
(42) = 07

c. (40) = CB
Result = (41) = CB
(42) = 00

d. (40) = 00
Result = (41) = 00
(42) = 00

**Flowchart:**



**Source Program:**

```
        SUB     A
        MOV     B,A     ;NUMBER OF SHIFTS = 0
        LXI     H,40H
        ADD     M       ;GET DATA
        JZ      DONE    ;THROUGH IF DATA 0
CHKMS:  JM      DONE    ;IS NEXT BIT 1?
        INR     B       ;NO, ADD 1 TO NUMBER OF SHIFTS
        ADD     A       ;SHIFT LEFT 1 BIT
        JMP     CHKMS
DONE:   INX     H       ;SAVE JUSTIFIED DATA
        MOV     M,A
        INX     H
        MOV     M,B     ;SAVE NUMBER OF SHIFTS
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | SUB | A | 97 |
| 01 | | MOV | B,A | 47 |
| 02 | | LXI | H,40H | 21 |
| 03 | | | | 40 |
| 04 | | | | 00 |
| 05 | | ADD | M | 86 |
| 06 | | JZ | DONE | CA |
| 07 | | | | 11 |
| 08 | | | | 00 |
| 09 | CHKMS: | JM | DONE | FA |
| 0A | | | | 11 |
| 0B | | | | 00 |
| 0C | | INR | B | 04 |
| 0D | | ADD | A | 87 |
| 0E | | JMP | CHKMS | C3 |
| 0F | | | | 09 |
| 10 | | | | 00 |
| 11 | DONE: | INX | H | 23 |
| 12 | | MOV | M,A | 77 |
| 13 | | INX | H | 23 |
| 14 | | MOV | M,B | 70 |
| 15 | HERE: | JMP | HERE | C3 |
| 16 | | | | 16 |
| 17 | | | | 00 |

JM DONE causes a Jump to location DONE if the Sign bit is 1. This condition may mean that the last result was a negative number or may just mean that its most significant bit was 1 — the computer only supplies the results; the programmer must provide the interpretation.

ADD A adds the number in the Accumulator to itself. The program uses this instruction rather than RAL or RLC, because ADD A affects the Sign bit while RAL and RLC do not.

## PROBLEMS

### 1) Checksum Of Data

**Purpose:** Calculate the checksum of a series of numbers. The length of the series is in memory location 41 and the series itself begins in memory location 42. Store the checksum in memory location 40. The checksum is formed by EXCLUSIVE-ORing all the numbers in the series into the Accumulator.

Note that such checksums are often used in paper tape and cassette systems to ensure that the data has been read correctly. The calculated checksum is compared to the one stored with the data — if the two checksums do not agree, the system will usually either indicate an error to the operator or automatically read the data again.

**Sample Problem:**

(41) = 03
(42) = 28
(43) = 55
(44) = 26

Result = (42) ⊕ (43) ⊕ (44)
= 28 ⊕ 55 ⊕ 26
= 00101000
⊕ 01010101
01111101
⊕ 00100110
01011011

= 5B

### 2) Sum Of 16-Bit Data

**Purpose:** Calculate the sum of a series of 16-bit numbers. The length of the series is in memory location 42 and the series itself begins in memory location 43. Store the sum in memory locations 40 and 41 (eight most significant bits in 41). Each 16-bit number occupies two memory locations with the eight most significant bits in the second one. Assume that the sum can be contained in 16 bits (i.e., no Carries).

**Sample Problem:**

(42) = 03
(43) = F1
(44) = 28
(45) = 1A
(46) = 30
(47) = 89
(48) = 4B
Result = 28F1 + 301A + 4B89 = A494
(40) = 94
(41) = A4

## 3) Number Of Zero, Positive, And Negative Numbers

**Purpose:** Determine the number of zero, positive (most significant bit zero but entire number not zero), and negative elements (most significant bit 1) in a block. The length of the block is in memory location 43 and the block itself starts in memory location 44. Place the number of negative elements in memory location 40, the number of zero elements in memory location 41, and the number of positive elements in memory location 42.

**Sample Problem:**

(43) = 06
(44) = 68
(45) = F2
(46) = 87
(47) = 00
(48) = 59
(49) = 2A

Result = 2 negative, 3 positive, and 1 zero so

(40) = 02
(41) = 01
(42) = 03

## 4) Find Minimum

**Purpose:** Find the smallest element in a block of data. The length of the block is in memory location 41 and the block itself begins in memory location 42. Store the minimum in memory location 40. Assume the numbers in the block are 8-bit unsigned binary numbers.

**Sample Problem:**

(41) = 05
(42) = 67
(43) = 79
(44) = 15
(45) = E3
(46) = 72

Result = (40) = 15 since this is the smallest of the five unsigned numbers.

## 5) Count 1 Bits

**Purpose:** Place the number of 1 bits in the contents of memory location 40 into memory location 41.

**Sample Problem:**

(40) = 3B = 0 0 1 1 1 0 1 1
Result = (41) = 05

# Chapter 6
# CHARACTER-CODED DATA

Microprocessors often handle character-coded data. Not only do keyboards, teletypewriters, communications devices, displays, and computer terminals expect or provide character-coded data — many instruments and control systems do also. The most commonly used code is ASCII. Baudot and EBCDIC are less frequently used. We will assume all of our character-coded data to be 7-bit ASCII with the most significant bit zero (see Table 6-1).

**HANDLING DATA IN ASCII**

Some principles to remember in handling ASCII-coded data are:

1) The codes for the numbers and letters form ordered subsequences. The decimal numbers are hex 30 through 39, the upper-case letters hex 41 through 5A. Thus you can do alphabetical ordering by sorting the data in increasing numerical order.
2) The computer draws no distinction between printing and nonprinting characters. This distinction is only made by the I/O device.
3) An ASCII device will interpret and send all data as ASCII. To print a 7 on an ASCII printer, the microprocessor must send hex 37; hex 07 is the 'bell' character. Similarly, the microprocessor will receive the character 9 from an ASCII keyboard as hex 39; hex 09 is the 'tab' character.
4) Some ASCII devices do not use the full character set. For example, control characters and lower-case letters may be ignored or printed as '?'.
5) Some widely used ASCII characters are:
   OA - line feed (LF)
   OD - carriage return (CR)
   20 - space
   3F - ?
   7F - rubout or delete character

Table 6-1. 7-Bit ASCII Character Code Chart

| Hex | ASCII | Hex | ASCII | Hex | ASCII |
|---|---|---|---|---|---|
| 00 | NUL | 2B | + | 56 | V |
| 01 | SOH | 2C | , | 57 | W |
| 02 | STX | 2D | — | 58 | X |
| 03 | ETX | 2E | . | 59 | Y |
| 04 | EOT | 2F | / | 5A | Z |
| 05 | ENQ | 30 | 0 | 5B | [ |
| 06 | ACK | 31 | 1 | 5C | \ |
| 07 | BEL | 32 | 2 | 5D | ] |
| 08 | BS | 33 | 3 | 5E | ^ (↑) |
| 09 | HT | 34 | 4 | 5F | — (←) |
| 0A | LF | 35 | 5 | 60 | ` |
| 0B | VT | 36 | 6 | 61 | a |
| 0C | FF | 37 | 7 | 62 | b |
| 0D | CR | 38 | 8 | 63 | c |
| 0E | SO | 39 | 9 | 64 | d |
| 0F | SI | 3A | : | 65 | e |
| 10 | DLE | 3B | ; | 66 | f |
| 11 | DC1 (X-ON) | 3C | < | 67 | g |
| 12 | DC2 (TAPE) | 3D | = | 68 | h |
| 13 | DC3 (X-OFF) | 3E | > | 69 | i |
| 14 | DC4 ~~(TAPE)~~ | 3F | ? | 6A | j |
| 15 | NAK | 40 | @ | 6B | k |
| 16 | SYN | 41 | A | 6C | l |
| 17 | ETB | 42 | B | 6D | m |
| 18 | CAN | 43 | C | 6E | n |
| 19 | EM | 44 | D | 6F | o |
| 1A | SUB | 45 | E | 70 | p |
| 1B | ESC | 46 | F | 71 | q |
| 1C | FS | 47 | G | 72 | r |
| 1D | GS | 48 | H | 73 | s |
| 1E | RS | 49 | I | 74 | t |
| 1F | US | 4A | J | 75 | u |
| 20 | SP | 4B | K | 76 | v |
| 21 | ! | 4C | L | 77 | w |
| 22 | " | 4D | M | 78 | x |
| 23 | # | 4E | N | 79 | y |
| 24 | $ | 4F | O | 7A | z |
| 25 | % | 50 | P | 7B | { |
| 26 | & | 51 | Q | 7C | \| |
| 27 | ' | 52 | R | 7D | } (ALT MODE) |
| 28 | ( | 53 | S | 7E | ~ |
| 29 | ) | 54 | T | 7F | DEL (RUB OUT) |
| 2A | * | 55 | U | | |

## EXAMPLES

## Length Of A String Of Characters

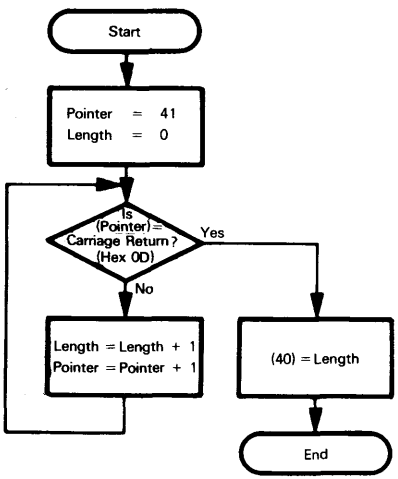
**Purpose:** Determine the length of a string of ASCII characters (7 bits with the most significant bit 0). The string starts in memory location 41; the end of the string is marked by a carriage return character ('CR', hex 0D). Place the length of the string (excluding the carriage return) in memory location 40.

**Sample Problems:**

a. (40) = 0D
Result = (40) = 00 since the first character is a carriage return.

b. (41) = 52 R
(42) = 41 A
(43) = 54 T
(44) = 48 H
(45) = 45 E
(46) = 52 R
(47) = 0D 'CR'
Result = (40) = 06

**Flowchart:**




**Source Program:**

```
        LXI     H,41H     ;POINTER = START OF STRING
        MVI     B,0       ;LENGTH = 0
        MVI     A,0DH
CHKCR:  CMP     M         ;IS CHARACTER 'CR'?
        JZ      DONE      ;YES, END OF STRING
        INR     B         ;NO, ADD 1 TO LENGTH
        INX     H
        JMP     CHKCR     ;EXAMINE NEXT CHARACTER
DONE:   MOV     A,B
        STA     40H       ;SAVE STRING LENGTH
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,41H | 21 |
| 01 | | | | 41 |
| 02 | | | | 00 |
| 03 | | MVI | B,0 | 06 |
| 04 | | | | 00 |
| 05 | | MVI | A,0DH | 3E |
| 06 | | | | 0D |
| 07 | CHKCR: | CMP | M | BE |
| 08 | | JZ | DONE | CA |
| 09 | | | | 10 |
| 0A | | | | 00 |
| 0B | | INR | B | 04 |
| 0C | | INX | H | 23 |
| 0D | | JMP | CHKCR | C3 |
| 0E | | | | 07 |
| 0F | | | | 00 |
| 10 | DONE: | MOV | A,B | 78 |
| 11 | | STA | 40H | 32 |
| 12 | | | | 40 |
| 13 | | | | 00 |
| 14 | HERE: | JMP | HERE | C3 |
| 15 | | | | 14 |
| 16 | | | | 00 |

The carriage return ('CR', 0D) is just another ASCII character as far as the computer is concerned. The fact that the I/O device treats 'CR' as a control character rather than as a printing character does not affect the computer.

The Compare instruction sets the flags but leaves the carriage return character in the Accumulator for later comparisons. The Zero flag is set as follows:

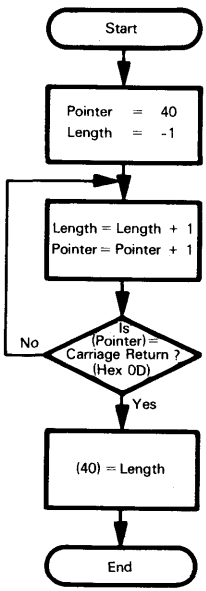
Zero = 1 if the character in the string is a carriage return
Zero = 0 if it is not a carriage return

The instruction INR B adds 1 to the string length counter in Register B. MVI B,0 initializes this counter to zero before the loop. Remember to initialize variables before using them in a loop.

This loop does not terminate because a counter decrements to zero. The computer will simply continue examining characters until it finds a carriage return. You may have to place a maximum count in a loop like this to avoid problems with erroneous strings which do not contain a carriage return. What would happen if the string did not contain a carriage return?

Note that by rearranging the logic and changing the initial conditions, you can eliminate several bytes of object code and decrease the execution time of the loop. If we adjust the flowchart so that the program increments the counter and pointer before it looks for the carriage return, only one Jump instruction is necessary instead of two. The new flowchart and program are:

**Flowchart:**




**Source Program:**

```
        LXI     H,40H       ;POINTER = BYTE BEFORE STRING
        MVI     B,0FFH      ;LENGTH = -1
        MVI     A,0DH
CHCKR:  INX     H
        INR     B           ;ADD 1 TO LENGTH
        CMP     M           ;IS CHARACTER 'CR'?
        JNZ     CHKCR       ;NO, KEEP COUNTING
        MOV     A,B
        STA     40H         ;YES, SAVE STRING LENGTH
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MVI | B,0FFH | 06 |
| 04 | | | | FF |
| 05 | | MVI | A,0DH | 3E |
| 06 | | | | 0D |
| 07 | NEXT: | INX | H | 23 |
| 08 | | INR | B | 04 |
| 09 | | CMP | M | BE |
| 0A | | JNZ | CHKCR | C2 |
| 0B | | | | 07 |
| 0C | | | | 00 |
| 0D | | MOV | A,B | 78 |
| 0E | | STA | 40H | 32 |
| 0F | | | | 40 |
| 10 | | | | 00 |
| 11 | HERE: | JMP | HERE | C3 |
| 12 | | | | 11 |
| 13 | | | | 00 |

## Find First Non-Blank Character

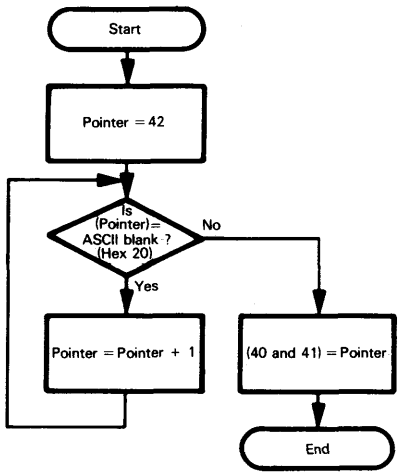
**Purpose:** Search a string of ASCII characters (7 bits with most significant bit 0) for a non-blank character. The string starts in memory location 42. Place the address of the first non-blank character in memory locations 40 and 41 (most significant bits in 41). A blank character is hex 20 in ASCII.

**Sample Problems:**

a. (42) = 37 (ASCII 7)
Result = (40) = 42
(41) = 00
since memory location 42 contains a non-blank character

b. (42) = 20 SP
(43) = 20 SP
(44) = 20 SP
(45) = 46 F
Result = (40) = 45
(41) = 00

**Flowchart:**




**Source Program:**

```
        LXI     H,42H     ;POINT TO START OF STRING
        MVI     A,20H     ;GET 'SP' FOR COMPARISON
CHBLK:  CMP     M         ;IS CHARACTER A BLANK?
        JNZ     DONE      ;NO, THROUGH
        INX     H
        JMP     CHBLK     ;YES, EXAMINE NEXT CHARACTER
DONE:   SHLD    40H       ;SAVE ADDRESS OF FIRST NON-BLANK CHARACTER
HERE:   JMP     HERE
```

**Object Program:**

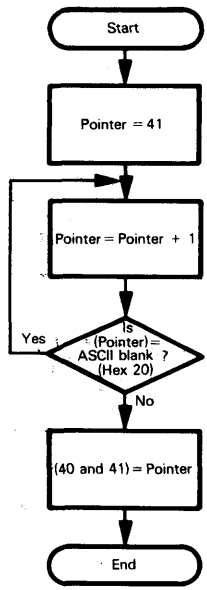
| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,42H | 21 |
| 01 | | | | 42 |
| 02 | | | | 00 |
| 03 | | MVI | A,20H | 3E |
| 04 | | | | 20 |
| 05 | CHBLK: | CMP | M | BE |
| 06 | | JNZ | DONE | C2 |
| 07 | | | | 0D |
| 08 | | | | 00 |
| 09 | | INX | H | 23 |
| 0A | | JMP | CHBLK | C3 |
| 0B | | | | 05 |
| 0C | | | | 00 |
| 0D | DONE: | SHLD | 40H | 22 |
| 0E | | | | 40 |
| 0F | | | | 00 |
| 10 | HERE: | JMP | HERE | C3 |
| 11 | | | | 10 |
| 12 | | | | 00 |

Looking for spaces in strings is a common task. Spaces frequently serve to separate fields, and often are eliminated from strings when they are used simply to increase readability or to fit particular formats. It is obviously wasteful to store and transmit beginning, ending, or extra spaces, particularly if you are paying for the communications capability and memory required.

The instruction SHLD is convenient for storing addresses in the 8080 format (least significant bits first). SHLD 40H stores the contents of Register L in memory location 40 and the contents of Register H in memory location 41.

Again, if we alter the initial conditions so that the loop control section precedes the processing section, we can reduce the number of bytes in the program and decrease the loop's execution time. The rearranged flowchart is:

**Flowchart:**




**Source Program:**

```
        LXI     H,41H   ;POINT TO BYTE BEFORE STRING
        MVI     A,20H   ;GET 'SP' FOR COMPARISON
NXTCH:  INX     H
        CMP     M       ;IS CHARACTER A BLANK?
        JZ      NXTCH   ;YES, KEEP EXAMINING CHARACTERS
        SHLD    40H     ;NO, SAVE ADDRESS OF FIRST NON-BLANK
                        ;CHARACTER
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,41H | 21 |
| 01 | | | | 41 |
| 02 | | | | 00 |
| 03 | | MVI | A,20H | 3E |
| 04 | | | | 20 |
| 05 | NXTCH: | INX | H | 23 |
| 06 | | CMP | M | BE |
| 07 | | JZ | NXTCH | CD |
| 08 | | | | 05 |
| 09 | | | | 00 |
| 0A | | SHLD | 40H | 22 |
| 0B | | | | 40 |
| 0C | | | | 00 |
| 0D | HERE: | JMP | HERE | C3 |
| 0E | | | | 0E |
| 0F | | | | 00 |

## Replace Leading Zeros With Blanks

**Purpose:** Edit a string of ASCII decimal characters so as to replace all leading zeros with blanks. The string starts in memory location 41; assume that it consists entirely of ASCII-coded decimal digits. The length of the string is in memory location 40.

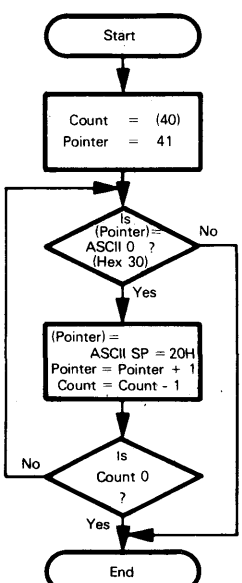
**Sample Problems:**

a.
(40) = 02
(41) = 36 = ASCII 6
Result = No change since the leading digit is not zero

b.
(40) = 08
(41) = 30 = ASCII 0
(42) = 30 = ASCII 0
(43) = 38 = ASCII 8
Result = (41) = 20 = ASCII SP
(42) = 20 = ASCII SP

The two leading ASCII zeros have been replaced by ASCII blanks.

**Flowchart:**



**Source Program:**

```
        LXI     H,40H
        MOV     B,M     ;COUNT = STRING LENGTH
        MVI     A,'0'   ;GET ASCII 0 FOR COMPARISON
CHKZ:   INX     H
        CMP     M       ;IS LEADING DIGIT 0?
        JNZ     DONE    ;NO, THROUGH
        MVI     M,20H   ;REPLACE LEADING ZERO WITH BLANK
        DCR     B       ;HAVE ALL DIGITS BEEN EXAMINED?
        JNZ     CHKZ    ;NO, GO EXAMINE NEXT DIGIT
DONE:   JMP     DONE
```

Single quotation marks around characters indicate ASCII.

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | MVI | A,'0' | 3E |
| 05 | | | | 30 |
| 06 | CHKZ: | INX | H | 23 |
| 07 | | CMP | M | BE |
| 08 | | JNZ | DONE | C2 |
| 09 | | | | 11 |
| 0A | | | | 00 |
| 0B | | MVI | M,20H | 36 |
| 0C | | | | 20 |
| 0D | | DCR | B | 05 |
| 0E | | JNZ | CHKZ | C2 |
| 0F | | | | 06 |
| 10 | | | | 00 |
| 11 | DONE: | JMP | DONE | C3 |
| 12 | | | | 11 |
| 13 | | | | 00 |

You will frequently want to edit decimal strings before they are printed or displayed to improve their appearance. Common editing tasks include eliminating leading zeros, justifying numbers, adding signs or other identifying markers, and rounding. Clearly, printed numbers like 0006 or $27.34382 can be confusing and annoying.

Here the loop has two exits — one if a processor finds a non-zero digit and the other if it has examined the entire string.

The instruction MVI M,20H places 20H in the memory location addressed by Registers H and L. You could also initialize Register C to 20H (i.e., MVI C,20H) and use MOV M,C to replace the leading zero with a blank. Note the tradeoffs involved in this example. MOV M,C executes faster than MVI M,20H and thus would decrease the inner loop's execution time. The overhead required, however, is an MVI C,20H in the initialization portion of the routine. If this example were to be used in a cash register application, which sequence would you choose and why?

All digits in the string are assumed to be in ASCII, i.e., the digits are hex 30 through 39 rather than the ordinary decimal 0 to 9. The conversion between decimal and ASCII is simply a matter of adding hex 30 to a decimal digit.

You may have to be careful to leave one zero in the event that all the digits are zero. How would you do this?

Note that each ASCII digit requires eight bits as compared to four for BCD. Therefore, ASCII is an expensive format in which to store or transmit numerical data.

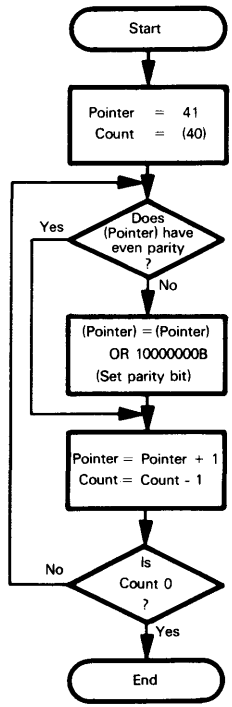
## Add Even Parity To ASCII Characters

**Purpose:** Add even parity to a string of 7-bit ASCII characters. The length of the string is in memory location 40 and the string itself begins in memory location 41. Place even parity in the most significant bit of each character by setting the most significant bit to 1 if that makes the total number of 1 bits in the word an even number.

**Sample Problem:**

(40) = 06
(41) = 31
(42) = 32
(43) = 33
(44) = 34
(45) = 35
(46) = 36
Result = (41) = B1
(42) = B3
(43) = 33
(44) = B4
(45) = 35
(46) = 36

**Flowchart:**




**Source Program:**

```
        LXI     H,40H
        MOV     B,M             ;GET STRING LENGTH
        MVI     C,10000000B     ;GET PARITY BIT OF 1
SETPR:  INX     H
        MOV     A,M             ;GET A CHARACTER
        ORA     C               ;SET PARITY BIT TO 1
        JPO     CHCNT           ;IS PARITY NOW EVEN?
        MOV     M,A             ;YES, SAVE CHARACTER WITH EVEN PARITY
CHCNT:  DCR     B
        JNZ     SETPR
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | MVI | C,10000000B | 0E |
| 05 | | | | 80 |
| 06 | SETPR: | INX | H | 23 |
| 07 | | MOV | A,M | 7E |
| 08 | | ORA | C | B1 |
| 09 | | JPO | CHCNT | E2 |
| 0A | | | | 0D |
| 0B | | | | 00 |
| 0C | | MOV | M,A | 77 |
| 0D | CHCNT: | DCR | B | 05 |
| 0E | | JNZ | SETPR | C2 |
| 0F | | | | 06 |
| 10 | | | | 00 |
| 11 | HERE: | JMP | HERE | C3 |
| 12 | | | | 11 |
| 13 | | | | 00 |

Parity is often added to ASCII characters before they are transmitted on noisy communication lines, providing a simple error checking facility. Parity will detect all single bit errors but does not allow error correction, i.e., you know that an error occurred when the received parity is wrong, but you cannot tell which bit was changed.

MVI C,10000000B saves a parity bit of 1 in Register C. (Note the use of the binary mask; the purpose of the mask is clearer when it is specified in this manner rather than as 80H or 128 decimal).

The instruction ORA C sets the parity (most significant) bit to 1 while retaining all the other bits as they were.

The following procedure is used to determine if the parity of the byte in memory is odd or even. We OR a parity bit into the byte loaded from memory; and then test to see if the parity is odd. If the parity is odd, then the byte already in memory has even parity, and we jump down to decrement the count of remaining bytes. If the parity is even, then we know that the byte in memory has odd parity, and therefore we store the byte in the Accumulator into that memory location.

The conditional jumps JPO (Jump On Parity Odd) and JPE (Jump On Parity Even) are seldom used except in parity generation and checking.

Do not confuse the Parity bit included in each character and the 8080's Parity bit, which is set to 1 if the last arithmetic or Boolean result had even parity.

## Pattern Match

**Purpose:** Compare two strings of ASCII characters to see if they are the same. The length of the strings is in memory location 41; one string starts in memory location 42 and the other in memory location 62. If the two strings match, clear memory location 40; otherwise, set memory location 40 to FF hex (all ones).

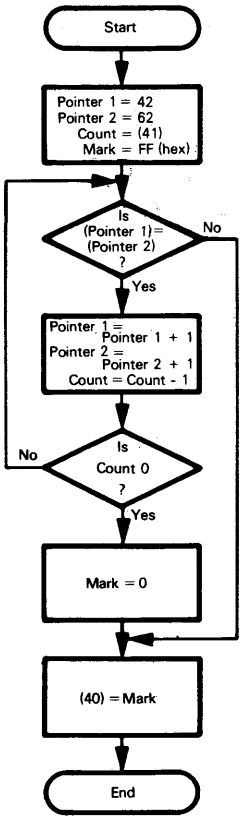
**Sample Problems:**

a.
(41) = 03
(42) = 43 C
(43) = 41 A
(44) = 54 T
(62) = 43 C
(63) = 41 A
(64) = 54 T
Result = (40) = 0 since the two strings are the same.

b.
(41) = 03
(42) = 52 R
(43) = 41 A
(44) = 54 T
(62) = 43 C
(63) = 41 A
(64) = 54 T
Result = (40) = FF (hex) since the first characters in the strings differ.

Note that the matching process ends as soon as the CPU finds a difference — the rest of the strings need not be examined.

**Flowchart:**




**Source Program:**

```
        LXI     H,41H
        MOV     B,M     ;COUNT = LENGTH OF STRINGS
        INX     H       ;POINTER 1 = START OF STRING 1
        LXI     D,62H   ;POINTER 2 = START OF STRING 2
        MVI     C,0FFH  ;MARK = FF (HEX)
CHCAR:  LDAX    D       ;GET CHARACTER FROM STRING 2
        CMP     M       ;IS THERE A MATCH?
        JNZ     DONE    ;NO, DONE
        INX     D
        INX     H
        DCR     B       ;HAVE ALL CHARACTERS BEEN CHECKED?
        JNZ     CHCAR   ;NO, CHECK NEXT PAIR
        MVI     C,0     ;YES, COMPLETE MATCH, MARK = 0
DONE:   MOV     A,C
        STA     40H     ;SAVE MARK
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,41H | 21 |
| 01 | | | | 41 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | INX | H | 23 |
| 05 | | LXI | D,62H | 11 |
| 06 | | | | 62 |
| 07 | | | | 00 |
| 08 | | MVI | C,0FFH | 0E |
| 09 | | | | FF |
| 0A | CHCAR: | LDAX | D | 1A |
| 0B | | CMP | M | BE |
| 0C | | JNZ | DONE | C2 |
| 0D | | | | 17 |
| 0E | | | | 00 |
| 0F | | INX | D | 13 |
| 10 | | INX | H | 23 |
| 11 | | DCR | B | 05 |
| 12 | | JNZ | CHCAR | C2 |
| 13 | | | | 0A |
| 14 | | | | 00 |
| 15 | | MVI | C,0 | 0E |
| 16 | | | | 00 |
| 17 | DONE: | MOV | A,C | 79 |
| 18 | | STA | 40H | 32 |
| 19 | | | | 40 |
| 1A | | | | 00 |
| 1B | HERE: | JMP | HERE | C3 |
| 1C | | | | 1B |
| 1D | | | | 00 |

Matching strings of ASCII characters is an essential part of looking for commands, recognizing names, identifying variables or operation codes in assemblers and compilers, finding files, and many other tasks.

The program uses two pointers, one in Registers H and L and the other in Registers D and E. The only instructions which use the address in D and E are LDAX (Load Accumulator Indirect) and STAX (Store Accumulator Indirect); arithmetic and logical operations with memory and transfers to other registers from memory (e.g., ADD M, AND M, MOV B,M, etc.) can only be performed using the address in Registers H and L.

The order of operations is very *important* because of the small number of instructions that use the address in Registers D and E. You must move a character from the string pointed to by D and E to the Accumulator and compare it to a character in the string pointed to by H and L. This order of operations is necessary because the 8080 has no instruction which allows a comparison to a character in a string pointed to by D and E.

For example, if you replaced LDAX D with MOV A,M, what would the next instruction be? This asymmetry is peculiar to the 8080 and can cause programming nightmares.

Note that each iteration updates both pointers.

This program could take advantage of the fact that a register is known to be zero after a particular conditional jump is executed. When the JNZ CHCAR instruction is executed, if the branch is not performed then we know that Register B contains zero. Therefore, we can move Register B to Register C, the Flag register, to indicate that a match has been found.

## PROBLEMS

### 1) Length Of A Teletypewriter Message

**Purpose:** Determine the length of an ASCII message. All characters are 7-bit ASCII with MSB 0. The string of characters in which the message is embedded starts in memory location 41. The message itself starts with an ASCII STX character (hex 02) and ends with ETX (hex 03). Place the length of the message (the number of characters between the STX and the ETX but including neither) in memory location 40.

**Sample Problem:**

(41) = 49
(42) = 02 STX
(43) = 47 G
(44) = 4F O
(45) = 03 ETX
Result = (40) = 02 since there are two characters between the STX in location 42 and the ETX in location 45.

### 2) Find Last Non-Blank Character

**Purpose:** Search a string of ASCII characters for the last non-blank character. The string starts in memory location 42 and ends with a carriage return character (hex 0D). Place the address of the last non-blank character in memory locations 40 and 41 (most significant bits in 41).

**Sample Problems:**

a.
(42) = 37 (ASCII 7)
(43) = 0D 'CR'
Result = (40) = 42
(41) = 00
since the last (and only) non-blank character is in memory location 42.

b.
(42) = 41 A
(43) = 20 'SP'
(44) = 48 H
(45) = 41 A
(46) = 54 T
(47) = 20 'SP'
(48) = 20 'SP'
(49) = 0D 'CR'
Result = (40) = 46
(41) = 00

## 3) Truncate Decimal String To Integer Form

**Purpose:** Edit a string of ASCII decimal characters so as to replace all digits to the right of the decimal point with ASCII blanks (hex 20). The string starts in memory location 41 and is assumed to consist entirely of ASCII-coded decimal digits and a possible decimal point (hex 2E). The length of the string is in memory location 40. If no decimal point appears in the string, assume that all digits are whole numbers with the decimal point (implicit) at the far right.

**Sample Problems:**

a.
(40) = 04
(41) = 37 7
(42) = 2E .
(43) = 38 8
(44) = 31 1
Result = (41) = 37 7
(42) = 2E .
(43) = 20 'SP'
(44) = 20 'SP'

b.
(40) = 03
(41) = 36 6
(42) = 37 7
(43) = 31 1
Result = Unchanged as number is assumed to be 671.

## 4) Check Even Parity In ASCII Characters

**Purpose:** Check even parity in a string of ASCII characters. The length of the string is in memory location 41 and the string itself begins in memory location 42. If the parity of all the characters in the string is correct, clear memory location 40; otherwise, set the contents of memory location 41 to FF hex (all ones).

**Sample Problems:**

a.
(41) = 03
(42) = B1
(43) = B2
(44) = 33
Result = (40) = 00 since all the characters have even parity.

b.
(41) = 03
(42) = B1
(43) = B6
(44) = 33
Result = (40) = FF (hex) since the character in memory location 42 does not have even parity.

## 5) String Comparison

**Purpose:** Compare two strings of ASCII characters to see which is larger (i.e., which follows the other in 'alphabetical' ordering). The length of the strings is in memory location 41; one string starts in memory location 42 and the other in memory location 62. If the string starting in memory location 42 is larger than or equal to the other string, clear memory location 40; otherwise, set memory location 40 to FF hex (all ones).

**Sample Problems:**

a.
(41) = 03
(42) = 43 C
(43) = 41 A
(44) = 54 T
(62) = 42 B
(63) = 41 A
(64) = 54 T
Result = (40) = 00 since CAT is 'larger' than BAT

b.
(41) = 03
(42) = 44 D
(43) = 4F O
(44) = 47 G
(62) = 44 D
(63) = 4F O
(64) = 47 G
Result = (40) = 00 since the two strings are equal.

c.
(41) = 03
(42) = 43 C
(43) = 41 A
(44) = 54 T
(62) = 43 C
(63) = 55 U
(64) = 54 T
Result = (40) = FF (hex) since CUT is 'larger' than CAT.

# Chapter 7
# CODE CONVERSION

Code conversion is a continual problem for microprocessors. Peripherals provide data in ASCII, BCD, or various special codes. The computer may be required to convert this data to binary or decimal in order to process it. Output devices require data in ASCII, BCD, 7-segment, or other codes. Therefore, the computer must convert the results to a suitable form after the processing is completed. Some code conversions are simple to perform in hardware; for example, standard integrated circuits exist for converting BCD to 7-segment. Universal Asynchronous Receiver/Transmitters (UARTs) convert between ASCII data and teletypewriter formats. However, the program may still be required to perform much of the conversion work.

## EXAMPLES

### Hex To ASCII

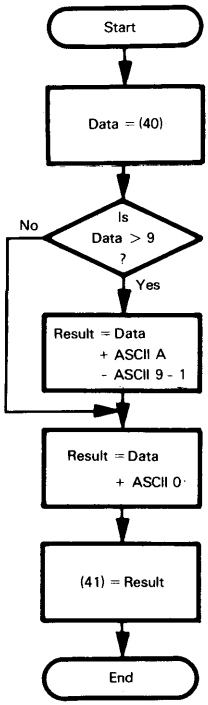
**Purpose:** Convert the contents of memory location 40 to an ASCII character. Memory location 40 contains a single hexadecimal digit (the four most significant bits are zero). Store the ASCII character in memory location 41.

**Sample Problems:**

a. (40) = 0C
Result = (41) = 43 'C'

b. (40) = 06
Result = (41) = 36 '6'

**Flowchart:**




**Source Program:**

```
        LDA     40H         ;GET DATA
        CPI     10          ;IS DATA 10 OR MORE?
        JNC     ASCZ
        ADI     'A'-'9'-1   ;YES, ADD OFFSET FOR LETTERS
ASCZ:   ADI     '0'         ;ADD OFFSET FOR ASCII
        STA     41H         ;STORE ASCII RESULT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LDA | 40H | 3A |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | CPI | 10 | FE |
| 04 | | | | 0A |
| 05 | | JNC | ASCZ | D2 |
| 06 | | | | 0A |
| 07 | | | | 00 |
| 08 | | ADI | 'A'-'9'-1 | C6 |
| 09 | | | | 07 |
| 0A | ASCZ: | ADI | '0' | C6 |
| 0B | | | | 30 |
| 0C | | STA | 41H | 32 |
| 0D | | | | 41 |
| 0E | | | | 00 |
| 0F | HERE: | JMP | HERE | C3 |
| 10 | | | | 0F |
| 11 | | | | 00 |

In this program, the basic idea is to add ASCII 0 to all the hexadecimal digits. This addition converts the decimal digits correctly; however, there is a break between ASCII 9 (39 hex) and ASCII A (41 hex) which must be taken into account. This break must be added to the nondecimal digits, i.e. A, B, C, D, E, F. This is accomplished by the ADI instruction which adds the offset 'A'-'9'-1 to the contents of the Accumulator. Can you explain why the offset is 'A'-'9'-1?

Note that the addition factors are placed in the assembly language program in ASCII form (single quotes surround ASCII characters). The offset for the letters is left as an arithmetic expression. The effort is to make the purpose of the factors as clear as possible in the assembly language listing. The extra assembly time is a very small price to pay for a large increase in clarity.

This routine could be used in a variety of programs; for example, monitor programs must convert hexadecimal digits to ASCII in order to display the contents of memory locations in hexadecimal on an ASCII printer or CRT display.

Another (quicker) conversion method that requires no conditional jumps at all is the following program described by Allison in COMPUTER magazine. (See Allison, D.R., "A Design Philosophy for Microcomputer Architectures", Computer, February 1977, pp. 35—41. This is an excellent article which we recommend highly.)

```
        LDA     40H         ;GET HEX DIGIT
        ADI     90H         ;DECIMAL ADD 90 BCD
        DAA
        ACI     40H         ;DECIMAL ADD 40 BCD + CARRY
        DAA
        STA     41H         ;STORE ASCII DIGIT
HERE:   JMP     HERE
```

Try this program on some digits. Can you explain why it works?

## Decimal To 7-Segment

**Purpose:** Convert the contents of memory location 40 to a 7-segment code in memory location 41. If memory location 40 does not contain a single decimal digit, clear memory location 41.

The following table can be used to convert decimal numbers to 7-segment code. The 7-segment code is organized with the most significant bit always zero followed by the code (1=on, 0=off) for segments g, f, e, d, c, b, a (see Figure 7-1).

| Digit | Code |
|---|---|
| 0 | 3F |
| 1 | 06 |
| 2 | 5B |
| 3 | 4F |
| 4 | 66 |
| 5 | 6D |
| 6 | 7D |
| 7 | 07 |
| 8 | 7F |
| 9 | 6F |

Note that the table uses 7D for 6 rather than the alternative (top bar off) 7C to avoid confusion with lower case b, and 6F for 9, rather than 67 (bottom bar off), for no particular reason.

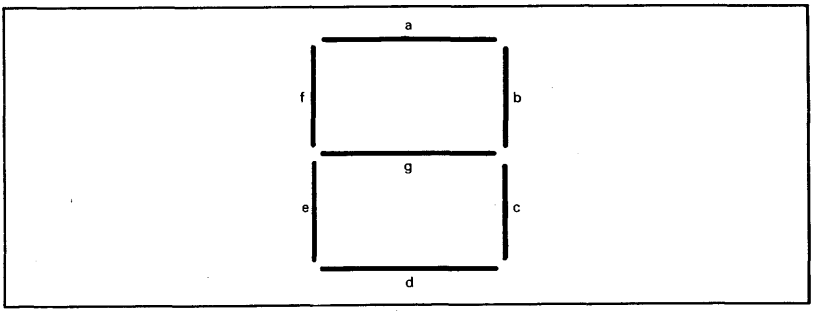


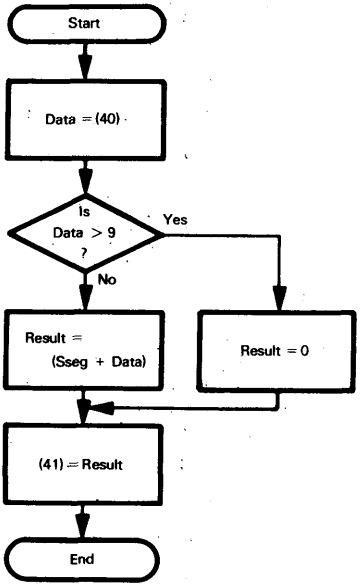

Figure 7-1. 7-Segment Arrangement

**Sample Problems:**

a. (40) = 03
Result = (41) = 4F

b. (40) = 28
Result = (41) = 00

**Flowchart:**




**Source Program:**

```
        MVI     B,0         ;GET ERROR CODE TO BLANK DISPLAY
        LDA     40H         ;GET DATA
        CPI     10          ;IS DATA > 9?
        JNC     DONE        ;YES, DONE
        LXI     H,SSEG      ;BASE ADDRESS OF 7-SEGMENT TABLE
                            ;MAKE DATA INTO A 16-BIT INDEX
        MOV     C,A
        DAD     B           ;FIND ELEMENT BY INDEXING
        MOV     B,M         ;GET 7-SEGMENT CODE
DONE:   MOV     A,B
        STA     41H         ;SAVE 7-SEGMENT CODE OR ZERO
                            ;FOR ERROR
HERE:   JMP     HERE
SSEG:   DB      3FH,06H,5BH,4FH,66H
        DB      6DH,7DH,07H,7FH,6FH
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | MVI | B,0 | 06 |
| 01 | | | | 00 |
| 02 | | LDA | 40H | 3A |
| 03 | | | | 40 |
| 04 | | | | 00 |
| 05 | | CPI | 10 | FE |
| 06 | | | | 0A |
| 07 | | JNC | DONE | D2 |
| 08 | | | | 10 |
| 09 | | | | 00 |
| 0A | | LXI | H,SSEG | 21 |
| 0B | | | | 17 |
| 0C | | | | 00 |
| 0D | | MOV | C,A | 4F |
| 0E | | DAD | B | 09 |
| 0F | | MOV | B,M | 46 |
| 10 | DONE: | MOV | A,B | 78 |
| 11 | | STA | 41H | 32 |
| 12 | | | | 41 |
| 13 | | | | 00 |
| 14 | HERE: | JMP | HERE | C3 |
| 15 | | | | 14 |
| 16 | | | | 00 |
| 17 | SSEG | DB | 3FH,06H,5BH,4FH,66H | 3F |
| 18 | | | | 06 |
| 19 | | | | 5B |
| 1A | | | | 4F |
| 1B | | | | 66 |
| 1C | | DB | 6DH,7DH,07H,7FH,6FH | 6D |
| 1D | | | | 7D |
| 1E | | | | 07 |
| 1F | | | | 7F |
| 20 | | | | 6F |

The program calculates the memory address of the desired code by adding the index (i.e., the digit to be displayed) to the base address of the seven-segment table. This procedure is known as a table look-up.

The assembly language pseudo-operation DB (DEFINE BYTE) is used to place constant data in program memory. Such data may include tables, headings, error messages, priming messages, format characters, thresholds, etc. The label attached to a DB pseudo-operation is assigned the value of the address into which the assembler places the first byte of data. The assembler simply places data for the table in memory. One DB pseudo-operation results in the filling of one or more memory locations.

Tables are often used to perform code conversion when the functional relationship is not as simple as that given under the HEX to ASCII example. Such tables usually just contain all the results organized according to the input data, i.e., the first entry is the code corresponding to the number zero.

Seven-segment code is commonly used in displaying digits and a few letters and other characters. Calculator-type seven-segment displays are inexpensive, easy to combine, and use little power; however, the seven-segment coded digits may be difficult to read.

## ASCII To Decimal

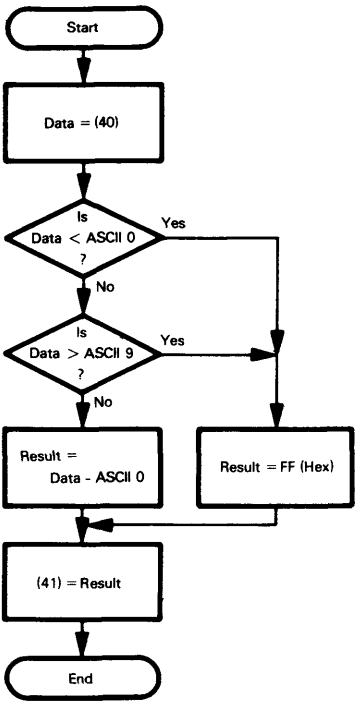
**Purpose:** Convert the contents of memory location 40 from an ASCII character to a decimal digit and store the result in memory location 41.. If the contents of memory location 40 is not the ASCII representation of a decimal digit, set the contents of memory location 41 to FF (hex).

**Sample Problems:**

a. (40) = 37 (ASCII 7)
Result = (41) = 07

b. (40) = 55
Result = (41) = FF

**Flowchart:**




**Source Program:**

```
        MVI     B,0FFH      ;GET ERROR MARKER (FF HEX)
        LDA     40H         ;GET DATA
        SUI     '0'         ;SUBTRACT ASCII 0
        JC      DONE        ;NOT DIGIT IF < ASCII 0
        CPI     10          ;IS RESULT ≥ 10?
        JNC     DONE        ;NOT DIGIT IF ≥ 10
        MOV     B,A
DONE:   MOV     A,B
        STA     41H         ;STORE DECIMAL RESULT OR ERROR MARKER
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | | MVI B,0FFH | 06 |
| 01 | | | FF |
| 02 | | LDA 40H | 3A |
| 03 | | | 40 |
| 04 | | | 00 |
| 05 | | SUI '0' | D6 |
| 06 | | | 30 |
| 07 | | JC DONE | DA |
| 08 | | | 10 |
| 09 | | | 00 |
| 0A | | CPI 10 | FE |
| 0B | | | 10 |
| 0C | | JNC DONE | D2 |
| 0D | | | 10 |
| 0E | | | 00 |
| 0F | | MOV B,A | 47 |
| 10 | DONE: | MOV A,B | 78 |
| 11 | | STA 41H | 32 |
| 12 | | | 41 |
| 13 | | | 00 |
| 14 | HERE: | JMP HERE | C3 |
| 15 | | | 14 |
| 16 | | | 00 |

This program determines if the data is between ASCII 0 and ASCII 9 inclusive. If so, the data is the ASCII representation of a decimal digit since the digits form a sequence. Subtracting hex 30 (ASCII 0) produces the decimal equivalent.

Note that one comparison is done with an actual subtraction (SUI '0') since that subtraction is necessary to convert ASCII to decimal. The second comparison is done with an implied subtraction (CPI 10) since the final decimal result is now in the Accumulator and we do not want to change it.

This kind of program is useful in a variety of situations, for example, ASCII to decimal conversion is necessary when decimal numbers are being entered from an ASCII device like a teletypewriter or CRT terminal.

## BCD To Binary

**Purpose:** Convert two BCD digits in memory locations 40 and 41 to a binary number in memory location 42. The most significant BCD digit is the one in memory location 40.

**Sample Problems:**

a. (40) = 02
(41) = 09
Result = (42) = 1D (hex) = 29 (decimal)

b. (40) = 07
(41) = 01
Result = (42) = 47 (hex) = 71 (decimal)

**Source Program:**

```
        LXI     H,40H
        MOV     A,M     ;GET MOST SIGNIFICANT DIGIT (MSD)
        ADD     A       ;MSD x 2
        MOV     B,A     ;SAVE MSD x 2
        ADD     A       ;MSD x 4
        ADD     A       ;MSD x 8
        ADD     B       ;MSD x 10
        INX     H       ;POINT TO LEAST SIGNIFICANT DIGIT
        ADD     M       ;ADD TO FORM BINARY EQUIVALENT
        INX     H
        MOV     M,A     ;STORE BINARY EQUIVALENT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | | LXI H,40H | 21 |
| 01 | | | 40 |
| 02 | | | 00 |
| 03 | | MOV A,M | 7E |
| 04 | | ADD A | 87 |
| 05 | | MOV B,A | 47 |
| 06 | | ADD A | 87 |
| 07 | | ADD A | 87 |
| 08 | | ADD B | 80 |
| 09 | | INX H | 23 |
| 0A | | ADD M | 86 |
| 0B | | INX H | 23 |
| 0C | | MOV M,A | 77 |
| 0D | HERE: | JMP HERE | C3 |
| 0E | | | 0D |
| 0F | | | 00 |

This program multiplies the BCD digit in memory location 40 by ten using repeated additions. Note that ADD A multiplies the contents of the Accumulator by 2. This allows you to multiply the Accumulator by small decimal numbers in a few instructions. How would you multiply the Accumulator by 16? by 12? by 7?

BCD entries are converted to binary in order to save on storage and calculations. Decimal numbers require additional memory space and more complex calculations. However, the conversion may offset some of the advantages of binary arithmetic.

BCD numbers require about 20% more memory than do binary numbers, e.g., 1000 requires 3 BCD digits (12 bits) and 10 bits in binary (since $2^{10} = 1024 \approx 1000$).

## ASCII String To Binary Number

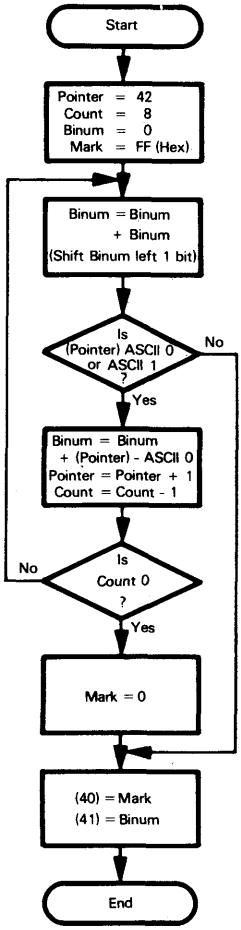
**Purpose:** Convert a string of 8 ASCII characters into a binary number and store the result in memory location 41. If any of the characters are not either ASCII zero or ASCII one, set memory location 40 to FF (hex); otherwise, clear memory location 40. The string of characters is in memory locations 42 through 49 with the most significant bit in memory location 42.

**Sample Problems:**

a.
(42) = 30 ('0')
(43) = 31 ('1')
(44) = 31 ('1')
(45) = 30 ('0')
(46) = 30 ('0')
(47) = 30 ('0')
(48) = 30 ('0')
(49) = 31 ('1')
Result = (40) = 00
(41) = 61 (i.e., 01100001)

b. Same as a. except:
(45) = 34 ('4')
Result = (40) = FF

**Flowchart:**



**Source Program:**

```
        LXI     H,42H     ;POINT TO START OF STRING
        SUB     A         ;BINARY NUMBER = 0
        MVI     B,8       ;COUNT = 8
        MVI     C,0FFH    ;MARKER = ERROR INDICATOR
BICON:  ADD     A         ;SHIFT BINARY NUMBER LEFT
        MOV     D,A
        MOV     A,M       ;GET CHARACTER FROM STRING
        SUI     '0'       ;CONVERT TO BINARY
        CPI     2         ;IS RESULT LESS THAN 2?
        JNC     DONE      ;NO, ERROR IN BINARY NUMBER
        ADD     D         ;ADD BIT TO BINARY NUMBER
        INX     H
        DCR     B         ;ALL CHARACTERS CONVERTED?
        JNZ     BICON
        MVI     C,0       ;YES, MARKER=ZERO, BINARY NUMBER
                          ;CORRECT
DONE:   LXI     H,40H     ;SAVE BINARY NUMBER
        MOV     A,M
        INX     H
        MOV     M,C       ;MARK IF CORRECT OR NOT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,42H | 21 |
| 01 | | | | 42 |
| 02 | | | | 00 |
| 03 | | SUB | A | 97 |
| 04 | | MVI | B,8 | 06 |
| 05 | | | | 08 |
| 06 | | MVI | C,0FFH | 0E |
| 07 | | | | FF |
| 08 | BICON: | ADD | A | 87 |
| 09 | | MOV | D,A | 57 |
| 0A | | MOV | A,M | 7E |
| 0B | | SUI | '0' | D6 |
| 0C | | | | 30 |
| 0D | | CPI | 2 | FE |
| 0E | | | | 02 |
| 0F | | JNC | DONE | D2 |
| 10 | | | | 1A |
| 11 | | | | 00 |
| 12 | | ADD | D | 82 |
| 13 | | INX | H | 23 |
| 14 | | DCR | B | 05 |
| 15 | | JNZ | BICON | C2 |
| 16 | | | | 08 |
| 17 | | | | 00 |
| 18 | | MVI | C,0 | 0E |
| 19 | | | | 00 |
| 1A | DONE: | LXI | H,40H | 21 |
| 1B | | | | 40 |
| 1C | | | | 00 |
| 1D | | MOV | A,M | 77 |
| 1E | | INX | H | 23 |
| 1F | | MOV | M,C | 71 |
| 20 | HERE: | JMP | HERE | C3 |
| 21 | | | | 20 |
| 22 | | | | 00 |

The instruction ADD A is a logical left shift, i.e., it moves the previously determined bits left one place and clears the least significant bit.

Since (B) = 0 when all the ASCII digits have been converted, we could replace MVI C,0 with MOV C,B (why?). The saving is one byte of code and a little execution time. Another alternative would be INR C (why?).

The conversion from ASCII simply involves subtracting ASCII 0 (hex 30).

The instruction CPI 2 sets

Carry = 1 if (A) $<$ 2, i.e., (A) = 0 or 1
Carry = 0 if (A) $\geq$ 2

The instruction ADD D adds the rest of the number being constructed into the Accumulator. Since the Accumulator contains a zero or a one and D contains a zero in the least significant bit, this instruction updates the number being constructed.

ASCII-to-binary conversion is necessary when numbers are entered in binary form from an ASCII device, as in an assembler which allows entries in binary.

## PROBLEMS

### 1) ASCII To Hex

**Purpose:** Convert the contents of memory location 40 to a hexadecimal digit and store the result in memory location 41. Assume that memory location 40 contains the ASCII representation of a hexadecimal digit (7 bits with MSB 0).

**Sample Problems:**

a. (40) = 43 'C'
Result = (41) = 0C

b. (40) = 36 '6'
Result = (41) = 06

### 2) 7-Segment To Decimal

**Purpose:** Convert the contents of memory location 40 from a 7-segment code to a decimal number in memory location 41. If memory location 40 does not contain a valid 7-segment code, set memory location 41 to FF (hex). Use the 7-segment table given under the Decimal To 7-Segment example and try to match codes.

**Sample Problems:**

a. (40) = 4F
Result = (41) = 03

b. (40) = 28
Result = (41) = FF

### 3) Decimal To ASCII

**Purpose:** Convert the contents of memory location 40 from a decimal digit to an ASCII character and store the result in memory location 41. If the number in memory location 40 is not a decimal digit, set the contents of memory location 41 to an ASCII blank character (20 hex).

**Sample Problem:**

a. (40) = 07
Result = (41) = 37 ('7')

b. (40) = 55
Result = (41) = 20 ('SP')

### 4) Binary to BCD

**Purpose:** Convert a binary number in memory location 40 to two BCD digits in memory locations 41 and 42 (most significant digit in 41). The number in memory location 40 is unsigned and less than 100.

**Sample Problems:**

a. (40) = 1D (29 decimal)
Result = (41) = 02
(42) = 09

b. (40) = 47 (71 decimal)
Result = (41) = 07
(42) = 01

## 5) Binary Number To ASCII String

**Purpose:** Convert the contents of memory location 41 to a string of ASCII characters representing the individual bits. The string occupies memory locations 42 through 49 with the most significant bit in memory location 42.

**Sample Problem:**

(41) = 61 (i.e., 01100001)
Result = (42) = 30 ('0')
(43) = 31 ('1')
(44) = 31 ('1')
(45) = 30 ('0')
(46) = 30 ('0')
(47) = 30 ('0')
(48) = 30 ('0')
(49) = 31 ('1')

# Chapter 8
# ARITHMETIC PROBLEMS

Most arithmetic in microprocessor applications consists of multiple-word binary or decimal manipulations. A decimal correction (Decimal Adjust) or some other means for performing decimal arithmetic is frequently the only arithmetic instruction beyond basic addition and subtraction. You must implement other arithmetic operations in software.

Multiple-precision binary arithmetic requires simple repetitions of the basic single-word instructions. The Carry bit transfers information between words. Add With Carry and Subtract With Borrow use the information from the previous arithmetic operation. You must be careful to clear the Carry before operating on the first words (obviously there is no Carry into or Borrow from the least significant bits).

Decimal arithmetic is a common enough task for microprocessors that most have a special instruction for this purpose. This instruction may either perform a decimal addition directly or correct the results of a binary addition to the proper decimal form. Decimal arithmetic is essential in such applications as point-of-sale terminals, calculators, check processors, order entry systems, and banking terminals.

You can implement multiplication and division as series of additions and subtractions respectively, much as they are done by hand. Double-word operations are important here since a multiplication produces a result twice as long as the operands while a division similarly contracts the length of the result. Multiplications and divisions are time-consuming when done in software because of the repeated arithmetic and shift operations that are necessary.

# EXAMPLES

## Multiple-Precision Addition

**Purpose:** Add two multiple-word binary numbers. The length of the numbers (in bytes) is in memory location 30, the numbers themselves start (least significant bits first) in memory locations 41 and 61 respectively, and the sum replaces the number starting in memory location 41.

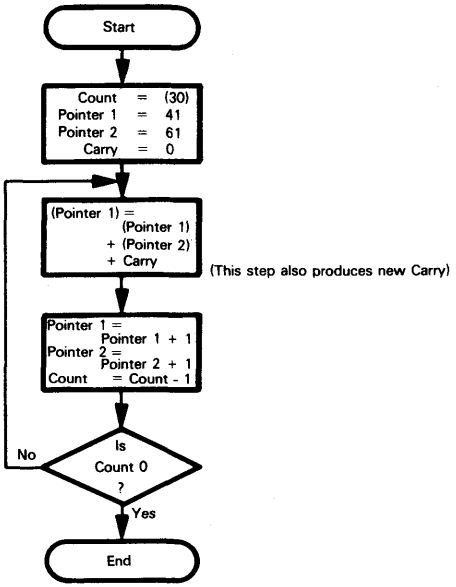
**Sample Problem:**

(30) = 04
(41) = C3
(42) = A7
(43) = 5B
(44) = 2F
(61) = B8
(62) = 35
(63) = DF
(64) = 14

Result = (41) = 7B
(42) = DD
(43) = 3A
(44) = 44

i.e., 2F5BA7C3
+14DF35B8
443ADD7B

**Flowchart:**


Start
Count = (30)
Pointer 1 = 41
Pointer 2 = 61
Carry = 0
(Pointer 1) = (Pointer 1) + (Pointer 2) + Carry
(This step also produces new Carry)
Pointer 1 = Pointer 1 + 1
Pointer 2 = Pointer 2 + 1
Count = Count - 1
Is Count 0 ?
No
Yes
End

**Source Program:**

```
        LDA     30H     ;COUNT=LENGTH OF STRINGS (IN BYTES)
        MOV     B,A
        LXI     H,41H   ;START POINTER 1 AT FIRST WORD OF STRING 1
        LXI     D,61H   ;START POINTER 2 AT FIRST WORD OF STRING 2
        ANA     A       ;CLEAR CARRY TO START
ADDW:   LDAX    D       ;GET WORD FROM STRING 2
        ADC     M       ;ADD WORDS FROM STRING 1
        MOV     M,A     ;STORE RESULT
        INX     D
        INX     H
        DCR     B
        JNZ     ADDW
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LDA | 30H | 3A |
| 01 | | | | 30 |
| 02 | | | | 00 |
| 03 | | MOV | B,A | 47 |
| 04 | | LXI | H,41H | 21 |
| 05 | | | | 41 |
| 06 | | | | 00 |
| 07 | | LXI | D,61H | 11 |
| 08 | | | | 61 |
| 09 | | | | 00 |
| 0A | | ANA | A | A7 |
| 0B | ADDW: | LDAX | D | 1A |
| 0C | | ADC | M | 8E |
| 0D | | MOV | M,A | 77 |
| 0E | | INX | D | 13 |
| 0F | | INX | H | 23 |
| 10 | | DCR | B | 05 |
| 11 | | JNZ | ADDW | C2 |
| 12 | | | | 0B |
| 13 | | | | 00 |
| 14 | HERE: | JMP | HERE | C3 |
| 15 | | | | 14 |
| 16 | | | | 00 |

The instruction ANA A is used to clear the Carry bit. Any other logical operation would have the same effect. The Carry must be cleared since there is no carry involved in the addition of the least significant bytes.

The instruction ADC, Add With Carry, includes the Carry from the previous words in the addition. ADC is the only instruction in the loop which affects the Carry. Remember that neither INX nor DCR affects the Carry.

Both the pointer in Registers D and E and the one in H and L must be updated during each iteration.

**DECIMAL ACCURACY IN BINARY**

This procedure can add binary numbers of any length. Note that ten binary bits correspond to three decimal digits since $2^{10} = 1024 \approx 1000$. So, you can calculate the number of bits required to give a certain accuracy in decimal digits. For example,

ten decimal digit accuracy requires:

$$10 \times \frac{10}{3} \approx 33 \text{ bits}$$

## Decimal Addition

**Purpose:** Add two multiple-word decimal (BCD) numbers. The length of the numbers is in memory location 30, the numbers themselves start (least significant bits first) in memory locations 41 and 61 respectively, and the sum replaces the number starting in memory location 41.

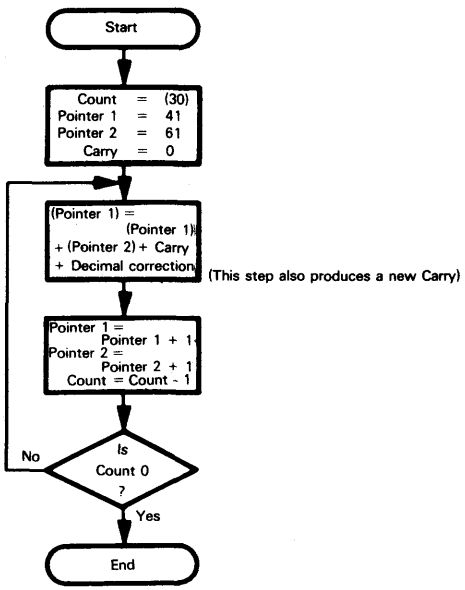
**Sample Problem:**

(30) = 04
(41) = 85
(42) = 19
(43) = 70
(44) = 36
(61) = 59
(62) = 34
(63) = 66
(64) = 12

Result = (41) = 44
(42) = 54
(43) = 36
(44) = 49

i.e., 36701985
+ 12663459
49365444

**Flowchart:**



**Source Program:**

```
        LDA     30H     ;COUNT=LENGTH OF STRINGS (IN BYTES)
        MOV     B,A
        LXI     H,41H   ;POINTER 1=FIRST WORD OF STRING 1
        LXI     D,61H   ;POINTER 2=FIRST WORD OF STRING 2
        ANA     A       ;CLEAR CARRY TO START
DECAD:  LDAX    D       ;GET 2 DIGITS FROM STRING 2
        ADC     M       ;ADD PAIR OF DIGITS FROM STRING 1
        DAA             ;MAKE ADDITIONAL DECIMAL
        MOV     M,A     ;STORE RESULT AS NEW STRING 1
        INX     D
        INX     H
        DCR     B
        JNZ     DECAD
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LDA | 30H | 3A |
| 01 | | | | 30 |
| 02 | | | | 00 |
| 03 | | MOV | B,A | 47 |
| 04 | | LXI | H,41H | 21 |
| 05 | | | | 41 |
| 06 | | | | 00 |
| 07 | | LXI | D,61H | 11 |
| 08 | | | | 61 |
| 09 | | | | 00 |
| 0A | | ANA | A | A7 |
| 0B | DECAD: | LDAX | D | 1A |
| 0C | | ADC | M | 8E |
| 0D | | DAA | | 27 |
| 0E | | MOV | M,A | 77 |
| 0F | | INX | D | 13 |
| 10 | | INX | H | 23 |
| 11 | | DCR | B | 05 |
| 12 | | JNZ | DECAD | C2 |
| 13 | | | | 0B |
| 14 | | | | 00 |
| 15 | HERE: | JMP | HERE | C3 |
| 16 | | | | 15 |
| 17 | | | | 00 |

The Decimal Adjust (DAA) instruction uses the Carry and Auxiliary Carry bits to correct the following situations:

**DECIMAL ADJUST**

1) The sum of two digits is between 10 and 15 inclusive. In this case, six must be added to the sum to give the right result, i.e.,

```
  0101   (5)
 +1000   (8)
  1101   (D)
 +0110
------
0011 0001  (BCD 13 which is correct)
```

2) The sum of two digits is 16 or more. In this case, the result is a proper BCD digit but six less than it should be, i.e.,

```
  1000   (8)
 +1001   (9)
------
0001 0001  (BCD 11)
 +0110
------
0001 0111  (BCD 17 which is correct)
```

Six must be added in both situations. However, case 1 can be recognized by the fact that the sum is not a BCD digit, i.e., it is between 10 and 15 (or A and F hexadecimal). Case 2 can only be recognized by the fact that the Carry (most significant digit) or Auxiliary Carry (least significant digit) has been set to 1 since the result is a valid BCD number. DAA is the only instruction which uses the Auxiliary Carry.

This procedure can add decimal (BCD) numbers of any length. Here four binary bits are required for each decimal digit so ten digit accuracy requires:

$$10 \times 4 = 40 \text{ bits}$$

as opposed to 33 bits in the binary case. This is essentially five 8-bit words instead of four. The decimal procedure also takes a little longer per word because of the extra DAA instruction.

## 8-Bit Binary Multiplication

**Purpose:** Multiply the 8-bit unsigned number in memory location 40 by the 8-bit unsigned number in memory location 41. Place the 8 least significant bits of the result in memory location 42 and the 8 most significant bits in memory location 43.

**Sample Problems:**

a.
```
(40) = 03
(41) = 05
Result = (42) = 0F
         (43) = 00
         i.e., 3 x 5 = 15
```

b.
```
(40) = 6F
(41) = 61
Result = (42) = 0F
         (43) = 2A
         i.e., 111 x 97 = 10,767
```

You can perform multiplication on a computer in the same way that you do long multiplication by hand. Since the numbers are binary, the only problem is whether to multiply by 0 or 1; multiplying by zero obviously gives zero as a result while multiplying by 1 produces the same number you started with. So each step in a binary multiplication can be reduced to the following operation:

**MULTIPLICATION ALGORITHM**

If the current bit in the multiplier is 1, add the multiplicand to the old product.

The only remaining problem is to ensure that you line everything up correctly each time. The following operations perform this task:

1) Shift multiplier left one bit so that the bit to be examined is placed in the Carry.
2) Shift product left one bit so that the next addition is lined up correctly.

The complete process for binary multiplication is as follows:

STEP 1 - Initialization:
PRODUCT = 0
COUNTER = 8

STEP 2 - Shift PRODUCT so as to line up properly:
PRODUCT = 2 x PRODUCT (LSB=0)

STEP 3 - Shift MULTIPLIER so bit goes to CARRY:
MULTIPLIER = 2 x MULTIPLIER

STEP 4 - Add MULTIPLICAND to PRODUCT if CARRY is 1:
If CARRY = 1, PRODUCT = PRODUCT + MULTIPLICAND

STEP 5 - Decrement COUNTER and check for zero:
COUNTER = COUNTER - 1
If COUNTER = 0, go to STEP 2

In the case of sample problem b, where the multiplier is 61 (hex) and the multiplicand is 6F (hex), the process works as follows:

Initialization:

| | |
|---|---|
| PRODUCT | 0000 |
| MULTIPLIER | 61 |
| MULTIPLICAND | 6F |
| COUNTER | 08 |

After First Iteration of STEPS 2 - 5:

| | |
|---|---|
| PRODUCT | 0000 |
| MULTIPLIER | C2 |
| MULTIPLICAND | 6F |
| COUNTER | 07 |
| CARRY FROM MULTIPLIER | 0 |

After Second Iteration:

| | |
|---|---|
| PRODUCT | 006F |
| MULTIPLIER | 84 |
| MULTIPLICAND | 6F |
| COUNTER | 06 |
| CARRY FROM MULTIPLIER | 1 |

After Third Iteration:

| | |
|---|---|
| PRODUCT | 014D |
| MULTIPLIER | 08 |
| MULTIPLICAND | 6F |
| COUNTER | 05 |
| CARRY FROM MULTIPLIER | 1 |

After Fourth Iteration:

| | |
|---|---|
| PRODUCT | 029A |
| MULTIPLIER | 10 |
| MULTIPLICAND | 6F |
| COUNTER | 04 |
| CARRY FROM MULTIPLIER | 0 |

After Fifth Iteration:

| | |
|---|---|
| PRODUCT | 0534 |
| MULTIPLIER | 20 |
| MULTIPLICAND | 6F |
| COUNTER | 03 |
| CARRY FROM MULTIPLIER | 0 |

After Sixth Iteration:

| | |
|---|---|
| PRODUCT | 0A68 |
| MULTIPLIER | 40 |
| MULTIPLICAND | 6F |
| COUNTER | 02 |
| CARRY FROM MULTIPLIER | 0 |

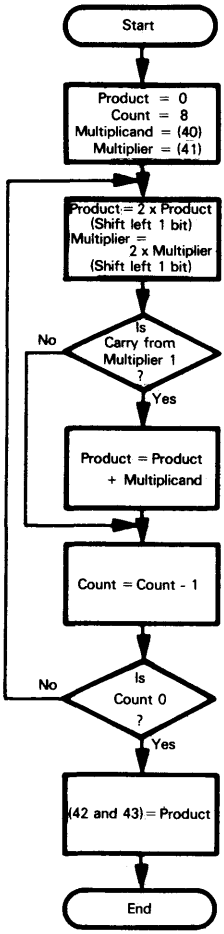
After Seventh Iteration:

| | |
|---|---|
| PRODUCT | 14D0 |
| MULTIPLIER | 80 |
| MULTIPLICAND | 6F |
| COUNTER | 01 |
| CARRY FROM MULTIPLIER | 0 |

After Eighth Iteration:

| | |
|---|---|
| PRODUCT | 2A0F |
| MULTIPLIER | 00 |
| MULTIPLICAND | 6F |
| COUNTER | 00 |
| CARRY FROM MULTIPLIER | 1 |

**Flowchart:**


Start
Product = 0
Count = 8
Multiplicand = (40)
Multiplier = (41)
Product = 2 x Product
(Shift left 1 bit)
Multiplier =
2 x Multiplier
(Shift left 1 bit)
Is
Carry from
Multiplier 1
?
No
Yes
Product = Product
+ Multiplicand
Count = Count - 1
Is
Count 0
?
No
Yes
(42 and 43) = Product
End

**Source Program:**

```
        LXI     H,40H
        MOV     E,M     ;GET MULTIPLICAND
        MVI     D,0     ;EXTEND TO 16 BITS
        INX     H
        MOV     A,M     ;GET MULTIPLIER
        LXI     H,0     ;PRODUCT = 0
        MVI     B,8     ;COUNT = 8
MULT:   DAD     H       ;PRODUCT = PRODUCT X 2
        RAL
        JNC     CHCNT   ;IS CARRY FROM MULTIPLIER 1?
        DAD     D       ;YES, PRODUCT = PRODUCT+MULTIPLICAND
CHCNT:  DCR     B
        JNZ     MULT
        SHLD    42H     ;SAVE PRODUCT IN MEMORY
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MOV | E,M | 5E |
| 04 | | MVI | D,0 | 16 |
| 05 | | | | 00 |
| 06 | | INX | H | 23 |
| 07 | | MOV | A,M | 7E |
| 08 | | LXI | H,0 | 21 |
| 09 | | | | 00 |
| 0A | | | | 00 |
| 0B | | MVI | B,8 | 06 |
| 0C | | | | 08 |
| 0D | MULT: | DAD | H | 29 |
| 0E | | RAL | | 17 |
| 0F | | JNC | CHCNT | D2 |
| 10 | | | | 13 |
| 11 | | | | 00 |
| 12 | | DAD | D | 19 |
| 13 | CHCNT: | DCR | B | 05 |
| 14 | | JNZ | MULT | C2 |
| 15 | | | | 0D |
| 16 | | | | 00 |
| 17 | | SHLD | | 22 |
| 18 | | | | 42 |
| 19 | | | | 00 |
| 1A | HERE: | JMP | HERE | C3 |
| 1B | | | | 1A |
| 1C | | | | 00 |

Note that the multiplicand must be extended to 16 bits by clearing Register D so that it can be added to the product using the DAD instruction.

The instruction DAD H acts as a 16-bit logical left shift for the 16-bit product.

In this program, the 8080 16-bit instructions handle data rather than addresses. LXI is used to initialize the product, DAD to perform 16-bit shifts and addition, and SHLD to store the result. You must be careful to extend 8-bit quantities (like the multiplicand) to 16 bits. Note that you cannot use the 16-bit facilities simultaneously for addressing and data manipulation.

Besides the obvious uses in calculators and point-of-sale terminals, multiplication is a key part of almost all signal processing and control algorithms. The speed at which multiplications can be performed may determine the usefulness of a CPU in process control, signal detection, and signal analysis.

This algorithm takes between 190 and 230 microseconds to multiply on an 8080 CPU with a 2 MHz clock. The precise time depends on the number of 1 bits in the multiplier. Other algorithms may be able to reduce the average execution time, but 200 microseconds will still be a typical execution time for a software multiplication.

## 8-Bit Binary Division

**Purpose:** Divide the 16-bit unsigned number in memory locations 40 and 41 (most significant bits in 41) by the 8-bit unsigned number in memory location 42. The numbers are normalized so that 1) the most significant bits of both the dividend and the divisor are zero and 2) the number in memory location 42 is greater than the number in memory location 41, i.e., the quotient can be contained in 8 bits. Store the quotient in memory location 43 and the remainder in memory location 44.

**Sample Problems:**

a.
(40) = 40 (64 decimal)
(41) = 00
(42) = 08
Result = (43) = 08
(44) = 00
i.e., 64/8 = 8

b.
(40) = 6D (12,909 decimal)
(41) = 32
(42) = 47 (71 decimal)
Result = (43) = B5 (181 decimal)
(44) = 3A (58 decimal)
i.e., 12,909/71 = 181 with a remainder of 58.

**DIVISION ALGORITHM**

You can perform division on the computer just like you would perform division with pen and paper, i.e., using trial subtractions. Since the numbers are binary, the only question is whether the bit in the quotient is 0 or 1, i.e., whether or not the divisor can be subtracted from what is left of the dividend. Each step in a binary division can be reduced to the following operation:

> If the divisor can be subtracted from the 8 most significant bits of the dividend without a Borrow, the corresponding bit in the quotient is 1; otherwise it is 0.

The only remaining problem is to line up the dividend and quotient properly. You can do this by shifting the dividend and quotient logically left one bit before each trial subtraction. The dividend and quotient can share a 16-bit register since the procedure clears one bit of the dividend at the same time as it determines one bit of the quotient.

The complete process for binary division is:

STEP 1 - Initialization:
QUOTIENT = 0
COUNT = 8

STEP 2 - Shift DIVIDEND and QUOTIENT so as to line up properly:
DIVIDEND = 2 x QUOTIENT
QUOTIENT = 2 x QUOTIENT

STEP 3 - Perform trial SUBTRACTION. If no BORROW add 1 to QUOTIENT:
If 8 MSBs of DIVIDEND $\geq$ DIVISOR then
MSBs of DIVIDEND = MSBs of DIVIDEND - DIVISOR
QUOTIENT = QUOTIENT+1

STEP 4 - Decrement counter and check for zero:
COUNT = COUNT - 1
If COUNT $\neq$ 0, go to STEP 2
REMAINDER = 8,MSBs of DIVIDEND

In the case of sample problem b, where the dividend is 326D (hex) and the divisor is 47 (hex), the process works as follows:

Initialization:

| | |
|---|---|
| DIVIDEND | 326D |
| DIVISOR | 47 |
| QUOTIENT | 00 |
| COUNT | 00 |

After first iteration of STEPS 2 - 4:
(Note that the dividend is shifted prior to the trial subtraction)

| | |
|---|---|
| DIVIDEND | 1DDA |
| DIVISOR | 47 |
| QUOTIENT | 01 |
| COUNT | 07 |

After second iteration of STEPS 2 - 4:

| | |
|---|---|
| DIVIDEND | 3BB4 |
| DIVISOR | 47 |
| QUOTIENT | 02 |
| COUNT | 06 |

After third iteration:

| | |
|---|---|
| DIVIDEND | 3068 |
| DIVISOR | 47 |
| QUOTIENT | 05 |
| COUNT | 05 |

After fourth iteration:

| | |
|---|---|
| DIVIDEND | 19D0 |
| DIVISOR | 47 |
| QUOTIENT | 0B |
| COUNT | 04 |

After fifth iteration:

| | |
|---|---|
| DIVIDEND | 33A0 |
| DIVISOR | 47 |
| QUOTIENT | 16 |
| COUNT | 03 |

After sixth iteration:

| | |
|---|---|
| DIVIDEND | 2040 |
| DIVISOR | 47 |
| QUOTIENT | 2D |
| COUNT | 02 |

After seventh iteration:

| | |
|---|---|
| DIVIDEND | 4080 |
| DIVISOR | 47 |
| QUOTIENT | 5A |
| COUNT | 01 |

After eighth iteration:

| | |
|---|---|
| DIVIDEND | 3A00 |
| DIVISOR | 47 |
| QUOTIENT | B5 |
| COUNT | 00 |

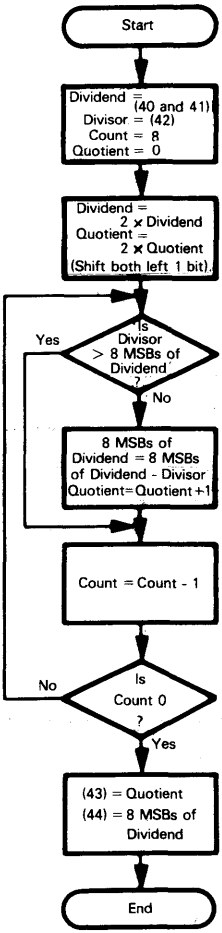
So the quotient is B5 and the remainder is 3A.

The MSBs of dividend and divisor are assumed to be zero so as to simplify calculations (the shift prior to the trial subtraction would otherwise place the MSB of the dividend in the Carry). Problems which do not have the required characteristics can be reformulated by removing those parts of the quotient which would overflow an 8-bit word. For example:

$$\frac{1024}{3} = \frac{400\ (\text{Hex})}{3} = 100 + \frac{100\ (\text{Hex})}{3}$$

The last problem is now in the proper form. An extra division may be necessary.

**Flowchart:**


Start
Dividend = (40 and 41)
Divisor = (42)
Count = 8
Quotient = 0
Dividend = 2 x Dividend
Quotient = 2 x Quotient
(Shift both left 1 bit).
Is Divisor > 8 MSBs of Dividend ?
Yes
No
8 MSBs of Dividend = 8 MSBs of Dividend - Divisor
Quotient=Quotient+1
Count = Count - 1
Is Count 0 ?
No
Yes
(43) = Quotient
(44) = 8 MSBs of Dividend
End

**Source Program:**

```
        LHLD    40H     ;GET DIVIDEND
        LDA     42H     ;GET DIVISOR
        MOV     C,A
        MVI     B,8     ;COUNT = 8
DIV:    DAD     H       ;SHIFT DIVIDEND, QUOTIENT
        MOV     A,H     ;IS MOST SIGNIFICANT PART OF DIVIDEND ≥ DIVISOR
        SUB     C
        JC      CNT     ;NO, GO TO NEXT STEP
        MOV     H,A     ;YES, SUBTRACT DIVISOR
        INR     L       ;AND ADD 1 TO QUOTIENT
CNT:    DCR     B
        JNZ     DIV
        SHLD    43H     ;STORE QUOTIENT, REMAINDER
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LHLD | 40H | 2A |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | LDA | 42H | 3A |
| 04 | | | | 42 |
| 05 | | | | 00 |
| 06 | | MOV | C,A | 4F |
| 07 | | MVI | B,8 | 06 |
| 08 | | | | 08 |
| 09 | DIV: | DAD | H | 29 |
| 0A | | MOV | A,H | 7C |
| 0B | | SUB | C | 91 |
| 0C | | JC | CNT | DA |
| 0D | | | | 11 |
| 0E | | | | 00 |
| 0F | | MOV | H,A | 67 |
| 10 | | INR | L | 2C |
| 11 | CNT: | DCR | B | 05 |
| 12 | | JNZ | DIV | C2 |
| 13 | | | | 09 |
| 14 | | | | 00 |
| 15 | | SHLD | 43H | 22 |
| 16 | | | | 43 |
| 17 | | | | 00 |
| 18 | HERE: | JMP | HERE | C3 |
| 19 | | | | 18 |
| 1A | | | | 00 |

Registers H and L hold both the dividend and the quotient. The quotient simply replaces the dividend in Register L as the dividend is shifted left logically.

An 8-bit subtraction is necessary (plus some register moves) since there is no simple way to do a 16-bit subtraction or comparison. However, DAD H provides a 16-bit logical left shift of the dividend and quotient.

The instruction INR L sets the least significant bit of the quotient to 1 since DAD H has previously cleared that bit.

Division is used in calculators, terminals, communications error checking, control algorithms, and many other applications.

The algorithm takes between 200 and 250 nanoseconds to divide on an Intel 8080 with a 2 MHz clock. The precise time depends on the number of 1 bits in the quotient. Other algorithms may reduce the average execution time, but 250 microseconds will still be typical for a software division.

## Self-Checking Numbers Double Add Double, Mod 10

**Purpose:** Calculate a checksum digit from a string of BCD digits. The length of the string of digits (number of words) is in memory location 30; the string of digits (2 BCD digits to a word) starts in memory location 41. Calculate the checksum digit by the DOUBLE ADD DOUBLE MOD 10 technique and store it in memory location 40. (See J.R. Herr, "Self-checking Number Systems", COMPUTER DESIGN, June 1974, pp. 85-91.)

The DOUBLE ADD DOUBLE MOD 10 technique works as follows:

1) Clear the checksum to start.
2) Multiply the leading digit by two and add the product to the checksum.
3) Add the next digit to the checksum.
4) Continue the alternating process until you have used all the digits.
5) The least significant digit of the checksum is the self-checking digit.

Self-checking digits are commonly added to identification numbers on credit cards, inventory tags, luggage, parcels, etc., which are handled by computerized systems. They may also be used in routing messages, identifying files, and other applications. The purpose of the digits is to minimize entry errors such as transposing digits (69 instead of 96), shifting digits (7260 instead of 3726), missing digits by 1 (65 instead of 64), etc. You can check the self-checking number automatically for correctness upon entry and can eliminate many errors immediately.

The analysis of self-checking methods is quite complex. For example, a plain checksum will not help with transposition (e.g., 4+9 = 9+4). The DOUBLE ADD DOUBLE algorithm finds transposition errors (e.g., $2 \times 4+9 = 17 \neq 2 \times 9+4$) but misses some errors (e.g., $2 \times 5+3$ has the same least significant digit as $2 \times 0+3$ so the method will not find that error).

For example, if the string of digits is:

549321

the result is:

Checksum = $5 \times 2+4+9 \times 2+3+2 \times 2+1 = 40$
Self-checking digit = 0 (least significant digit of checksum)

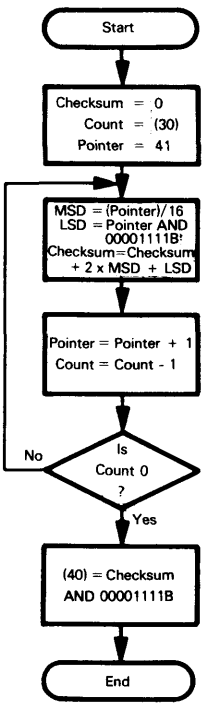
Note that an erroneous entry like 543921 would produce a different self-checking digit (4) but an erroneous entry like 043921 would be undetectable.

**Sample Problems:**

a.
(30) = 03
(41) = 36
(42) = 68
(43) = 51
Result = Checksum = 3 x 2+6+6 x 2+8+5 x 2+1 = 43
(40) = 03

b.
(30) = 04
(41) = 50
(42) = 29
(43) = 16
(44) = 83
Result = Checksum = 5 x 2+0+2 x 2+9+1 x 2+6+8 x 2+3 = 50
(40) = 00

**Flowchart:**



**Source Program:**

```
        LDA     30H             ;COUNT=LENGTH OF STRING (IN BYTES)
        MOV     B,A
        MVI     C,0             ;CHECKSUM=0
        LXI     H,41H           ;POINT TO START OF STRING
CHDIG:  MOV     A,M             ;GET TWO BCD DIGITS
        MOV     D,A             ;SAVE COPY
        RAR                     ;GET MSD BY SHIFT AND MASK
        RAR
        RAR
        RAR
        ANI     00001111B       ;MASK OFF MSD
        ADD     A               ;DOUBLE MSD
        DAA                     ;KEEP IT DECIMAL
        ADD     C               ;ADD 2 x MSD TO CHECKSUM
        DAA
        MOV     C,A
        MOV     A,D
        ANI     00001111B       ;MASK OFF LSD
        ADD     C               ;ADD LSD TO CHECKSUM
        DAA
        MOV     C,A
        INX     H
        DCR     B
        JNZ     CHDIG
        ANI     00001111B       ;MASK OFF SELF-CHECKING DIGIT
        STA     40H             ;SAVE SELF-CHECKING DIGIT
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LDA | 30H | 3A |
| 01 | | | | 30 |
| 02 | | | | 00 |
| 03 | | MOV | B,A | 47 |
| 04 | | MVI | C,0 | 0E |
| 05 | | | | 00 |
| 06 | | LXI | H,41H | 21 |
| 07 | | | | 41 |
| 08 | | | | 00 |
| 09 | CHDIG: | MOV | A,M | 7E |
| 0A | | MOV | D,A | 57 |
| 0B | | RAR | | 1F |
| 0C | | RAR | | 1F |
| 0D | | RAR | | 1F |
| 0E | | RAR | | 1F |
| 0F | | ANI | 00001111B | E6 |
| 10 | | | | 0F |
| 11 | | ADD | A | 87 |
| 12 | | DAA | | 27 |
| 13 | | ADD | C | 81 |
| 14 | | DAA | | 27 |
| 15 | | MOV | C,A | 4F |
| 16 | | MOV | A,D | 7A |
| 17 | | ANI | 00001111B | E6 |
| 18 | | | | 0F |
| 19 | | ADD | C | 81 |
| 1A | | DAA | | 27 |
| 1B | | MOV | C,A | 4F |
| 1C | | INX | H | 23 |
| 1D | | DCR | B | 05 |
| 1E | | JNZ | CHDIG | C2 |
| 1F | | | | 09 |
| 20 | | | | 00 |
| 21 | | ANI | 00001111B | E6 |
| 22 | | | | 0F |
| 23 | | STA | 40H | 32 |
| 24 | | | | 40 |
| 25 | | | | 00 |
| 26 | HERE: | JMP | HERE | C3 |
| 27 | | | | 26 |
| 28 | | | | 00 |

The digits are removed by shifting and masking. Four right shifts are needed to get the most significant digit.

A Decimal Adjust (DAA) must follow each addition to get the proper decimal result. A single DAA after a series of additions will not do the job (try it in this program).

Doubling the MSD involves adding it to itself and then performing the Decimal Adjust (DAA).

Carries from the decimal sum are ignored since the procedure only uses the least significant digit of the checksum.

## PROBLEMS

### 1) Multiple-Precision Subtraction

**Purpose:** Subtract two multiple-word numbers. The length of the numbers is in memory location 30, the numbers themselves start (least significant bits first) in memory locations 41 and 61 respectively, and the difference replaces the number starting in memory location 41. Subtract the number starting in 61 from the one starting in 41.

**Sample Problem:**

(30) = 04
(41) = C3
(42) = A7
(43) = 5B
(44) = 2F
(61) = B8
(62) = 35
(63) = 0F
(64) = 14

Result = (41) = 0B
(42) = 72
(43) = 7C
(44) = 1A

i.e.
```
  2F5BA7C3
+14 0F 35 B8
  1A7C720B
```

### 2) Decimal Subtraction

**Purpose:** Subtract two multiple-word decimal (BCD) numbers. The length of the numbers is in memory location 30, the numbers themselves start (least significant bits first) in memory locations 41 and 61 respectively and the difference replaces the number starting in 61 from the one starting in 41.

**Sample Problem:**

(30) = 04
(41) = 85
(42) = 19
(43) = 70
(44) = 36
(61) = 59
(62) = 34
(63) = 66
(64) = 12

Result = (41) = 26
(42) = 85
(43) = 03
(44) = 24

i.e.
```
  36701985
+12663459
  24038526
```

This will serve as a hint:

$$X - Y = X + 99 - Y + \overline{BORROW}$$

where X and Y are each 2 digits from the strings and the BORROW is the borrow from the less significant digits. Finding 99-Y is no problem since Y is always less than or equal to 99. But remember that the role of the CARRY is reversed from the usual (why?).

## 3) 8-Bit By 16-Bit Binary Multiplication

**Purpose:** Multiply the 16-bit unsigned number in memory locations 40 and 41 (most significant bits in 41) by the 8-bit unsigned number in memory location 42. Store the result in memory locations 43 through 45 with the most significant bits in memory location 45.

**Sample Problems:**

a.
(40) = 03
(41) = 00
(42) = 05
Result = (43) = 0F
(44) = 00
(45) = 00
i.e., 3 x 5 = 15

b.
(40) = 6F (29,295 decimal)
(41) = 72
(42) = 61 (97 decimal)
Result = (43) = 0F
(44) = 5C
(45) = 2B
i.e., 29,295 x 97 = 2,841,615

Proceed just as in the example under 8-Bit Binary Multiplication but save the 8 most significant bits of the product in the Accumulator, i.e., shift them in as the multiplier is shifted out.

## 4) Signed Binary Division

**Purpose:** Divide the 16-bit signed number in memory locations 40 and 41 (most significant bits in 41) by the 8-bit signed number in memory location 42. The numbers are normalized so that the magnitude of memory location 42 is greater than the magnitude of memory location 41, i.e., the quotient can be contained in 8 bits. Store the quotient (signed) in memory location 43 and the remainder (always positive) in memory location 44.

**Sample Problems:**

a.
(40) = C0 (-64)
(41) = FF
(42) = 08
Result = (43) = F8 (-8) quotient
(44) = 00 (0) remainder

b.
(40) = 93 (-4,717)
(41) = ED
(42) = 47 (71) decimal
Result = (43) = BD (-67) decimal
(44) = 2B (+40) decimal

Determine the sign of the result, perform an unsigned division, and adjust the quotient and remainder properly.

## 5) Self-Checking Numbers ALIGNED 1, 3, 7 MOD 10

**Purpose:** Calculate a checksum digit from a string of BCD digits. The length of the string of digits (number of words) is in memory location 30; the string of digits (2 BCD digits to a word) starts in memory location 41. Calculate the checksum digit by the ALIGNED 1, 3, 7 MOD 10 method and store it in memory location 40.

The ALIGNED 1, 3, 7 MOD 10 technique works as follows:

1) Clear the checksum to start.
2) Add the leading digit to the checksum.
3) Multiply the next digit by 3 and add the product to the checksum.
4) Multiply the next digit by 7 and add the product to the checksum.
5) Continue the process (Steps 2-4) until you have used all the digits.
6) The self-checking digit is the least significant of the checksum.

For example, if the string of digits is:

549321

the result is:

Checksum = $5 + 3 \times 4 + 7 \times 9 + 3 + 3 \times 2 + 7 \times 1 = 96$
Self-checking digit = 6

**Sample Problems:**

a. (30) = 03
(41) = 36
(42) = 68
(43) = 51
Result = Checksum = $3 + 3 \times 6 + 7 \times 6 + 8 + 3 \times 5 + 7 \times 1 = 93$
(40) = 03

b. (30) = 04
(41) = 50
(42) = 29
(43) = 16
(44) = 83
Result = Checksum = $5 + 3 \times 0 + 7 \times 2 + 9 + 3 \times 1 + 7 \times 6 + 8 + 3 \times 3 = 90$
(40) = 00

Note that $7 = 2 \times 3 + 1$ and $3 = 2 \times 1 + 1$ so the formula $M_i = 2 \times M_{i-1} + 1$ can be used to get from one multiplying factor to the next.

# Chapter 9
# TABLES AND LISTS

Tables and lists are two of the basic data structures used with all computer systems. We have already seen tables used to perform arithmetic and code conversions. Tables may also identify or respond to commands and instructions, linearize data, provide access to files or records, define the meaning of the keys or switches, and choose instruction sequences. Lists are usually less structured than tables. Lists may record tasks which the processors must perform, output messages or data which the processor must record, or conditions which have changed or should be monitored. Tables are a simple way of making decisions or solving problems since no computations or logical functions are necessary. The problem is simply one of organizing the table so that the proper entry is easy to find. Lists allow the execution of multiple tasks, the preparation of multiple results, and the construction of interrelated data files (or data bases). Problems include how to add elements to the list and delete elements from it.

## EXAMPLES

### Add Entry To List

**Purpose:** Add the contents of memory location 30 to a list if it is not already present. The length of the list is in memory location 40 and the list itself begins in memory location 42.

**Sample Problems:**

a.

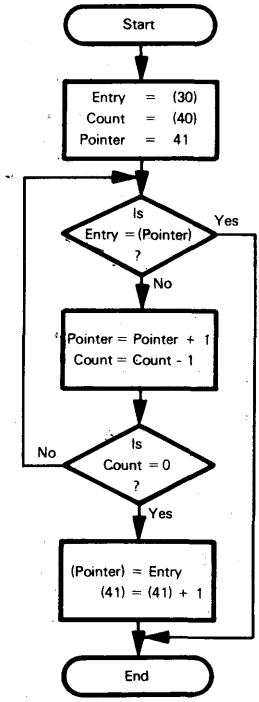
(30) = 6B
(40) = 04
(41) = 37
(42) = 61
(43) = 28
(44) = 1D
Result = (40) = 05
(45) = 6B

The entry is added to the list since it is not already present. The length of the list is increased by 1.

b.

(30) = 6B
(40) = 04
(41) = 37
(42) = 6B
(43) = 28
(44) = 1D
Result = No change since the entry is already in the list.

**Flowchart:**




**Source Program:**

```
        LXI     41H
        MOV     B,M     ;COUNT = LENGTH OF LIST
        INX     H       ;POINT TO START OF LIST
        LDA     30H     ;GET ENTRY
SRLST:  CMP     M       ;IS ENTRY = ELEMENT IN LIST?
        JZ      DONE    ;YES, THROUGH
        INX     H       ;NO, GO ON TO NEXT ELEMENT
        DCR     B
        JNZ     SRLST   ;HAVE ALL ELEMENTS BEEN EXAMINED?
        MOV     M,A     ;YES, ADD ENTRY TO LIST
        LXI     H,41H
        INR     M       ;ADD 1 TO LIST LENGTH
DONE:   JMP     DONE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,41H | 21 |
| 01 | | | | 41 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | INX | H | 23 |
| 05 | | LDA | 30H | 3A |
| 06 | | | | 30 |
| 07 | | | | 00 |
| 08 | SRLST: | CMP | M | BE |
| 09 | | JZ | DONE | CA |
| 0A | | | | 16 |
| 0B | | | | 00 |
| 0C | | INX | H | 23 |
| 0D | | DCR | B | 05 |
| 0E | | JNZ | SRLST | C2 |
| 0F | | | | 08 |
| 10 | | | | 00 |
| 11 | | MOV | M,A | 77 |
| 12 | | LXI | H,41H | 21 |
| 13 | | | | 41 |
| 14 | | | | 00 |
| 15 | | INR | M | 34 |
| 16 | DONE: | JMP | DONE | C3 |
| 17 | | | | 16 |
| 18 | | | | 00 |

The program does not work if the length of the list could be zero. We could avoid this problem by checking the length initially. The initialization procedure would then be:

```
        LXI     H,42H
        MOV     B,M     ;COUNT = LENGTH OF LIST
        SUB     A
        ORA     B       ;SET FLAGS FROM COUNT
        LDA     30H     ;GET ENTRY
        INX     H       ;POINT TO START OF LIST
        JZ      ADELM   ;ADD 1ST ENTRY IF LIST EMPTY
        -
        -
        -
ADELM:  MOV     M,A     ;ADD ENTRY TO LIST
        -
        -
        -
```

The procedure:

```
        LXI     H,ADDR
        INR     M
```

is a quick way to add 1 to a counter in memory location ADDR. Using DCR M in a similar fashion subtracts 1 from the counter. MVI M,CONST can place a starting value (such as zero) in the counter. Memory locations should, of course, only be used for counters when no registers are available.

Clearly this method of adding elements is very inefficient if the list is long. We could improve the procedure by limiting the search to part of the list or by ordering the list. We could limit the search by using the entry to get a starting point in the list. This method is called hashing and is much like selecting a starting page in a dictionary or directory on the basis of the first letter in an entry. We could order the list by numerical value. The search could then end when the list values went beyond the entry (the values would be either larger or smaller depending on the ordering technique used). The difficulty with this method is that a new entry would have to be inserted properly and then all the other entries would have to be moved down in the list.

The program could be restructured to use two tables. One table could provide a starting search point in the other table, e.g., the search point could be based on the most or least significant 4-bit digit in the entry.

If each entry were longer than one word, a pattern-matching procedure would be necessary as in the example given under PATTERN MATCH. We would have to be careful to line up the next entry correctly if a match failed, i.e., skip over the last part of the present entry once a mismatch was found.

## Check An Ordered List

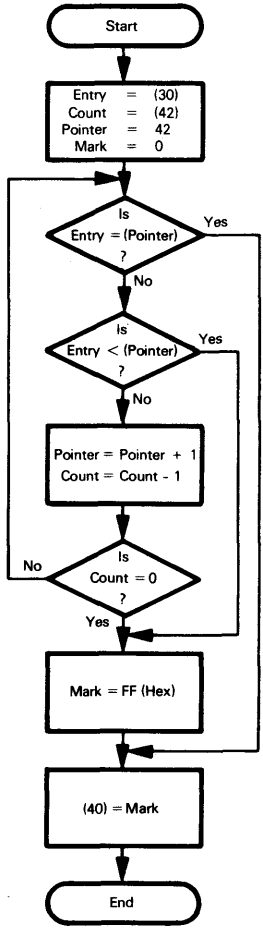
**Purpose:** Check the contents of memory location 30 to see if it is in an ordered list. The length of the list is in memory location 41; the list itself begins in memory location 42 and consists of unsigned binary numbers in increasing order. If the contents of location 30 is in the list, clear memory location 40; otherwise, set memory location 40 to FF (hex).

**Sample Problems:**

a.
(30) = 6B
(41) = 04
(42) = 37
(43) = 55
(44) = 7D
(45) = A1
Result = (40) = FF

b.
(30) = 6B
(41) = 04
(42) = 37
(43) = 55
(44) = 6B
(45) = A1
Result = (40) = 00

**Flowchart:**


Start
Entry = (30)
Count = (42)
Pointer = 42
Mark = 0
Is Entry = (Pointer) ?
Yes
No
Is Entry < (Pointer) ?
Yes
No
Pointer = Pointer + 1
Count = Count - 1
Is Count = 0 ?
No
Yes
Mark = FF (Hex)
(40) = Mark
End

**Source Program:**

```
        LXI     H,41H
        MOV     B,M     ;COUNT = LENGTH OF LIST
        INX     H       ;POINT TO START OF LIST
        MVI     C,0     ;MARK = 0 FOR IN LIST
        LDA     30H     ;GET ENTRY
SRLST:  CMP     M       ;IS ENTRY = ELEMENT IN LIST?
        JZ      DONE    ;YES, THROUGH
        JC      NOTIN   ;ENTRY NOT IN LIST IF < ELEMENT
        INX     H       ;GO ON TO NEXT ELEMENT
        DCR     B
        JNZ     SRLST   ;ALL ELEMENTS EXAMINED?
NOTIN:  MVI     C,0FFH  ;YES, MARK = FF FOR NOT IN LIST
DONE:   MOV     A,C     ;SAVE MARK = 0 OR FF
        STA     40H
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,41H | 21 |
| 01 | | | | 41 |
| 02 | | | | 00 |
| 03 | | MOV | B,M | 46 |
| 04 | | INX | H | 23 |
| 05 | | MVI | C,0 | 0E |
| 06 | | | | 00 |
| 07 | | LDA | 30H | 3A |
| 08 | | | | 30 |
| 09 | | | | 00 |
| 0A | SRLST: | CMP | M | BE |
| 0B | | JZ | DONE | CA |
| 0C | | | | 18 |
| 0D | | | | 00 |
| 0E | | JC | NOTIN | DA |
| 0F | | | | 16 |
| 10 | | | | 00 |
| 11 | | INX | H | 23 |
| 12 | | DCR | B | 05 |
| 13 | | JNZ | SRLST | C2 |
| 14 | | | | 0C |
| 15 | | | | 00 |
| 16 | NOTIN: | MVI | C,0FFH | 0E |
| 17 | | | | FF |
| 18 | DONE: | MOV | A,C | 79 |
| 19 | | STA | 40H | 32 |
| 1A | | | | 40 |
| 1B | | | | 00 |
| 1C | HERE: | JMP | HERE | C3 |
| 1D | | | | 1C |
| 1E | | | | 00 |

The searching process is a bit different here since the elements are ordered. Once we find an element larger than the entry, the search is over since subsequent elements will be even larger. You may want to try an example to convince yourself that the procedure works.

**SEARCHING METHODS**

As in the previous problem, a table or other method which chose a good starting point would speed the search. One method would be to start in the middle and then determine which half of the list the entry was in, then divide the half into halves, etc. This method is called a "binary search" since it divides the remaining part of the list in half each time. Knuth describes other searching techniques in his book "The Art Of Computer Programming, Volume III: Sorting and Searching", Addison-Wesley, Reading, Mass., 1973. Knuth also has discussed searching and hashing in a more elementary way in an article entitled "Algorithms" (see the April 1977 issue of Scientific American).

This algorithm is a bit slower than the one in the example given under ADD ENTRY TO LIST because of the extra conditional Jump (JC NOTIN). The average execution time for this simple search technique increases linearly with the length of the list, while the average execution time for a binary search is logarithmically related to the length of the list (the search time increases by one iteration time when the length doubles). Note that we could replace the MVI C,0FFH instruction at label NOTIN with a DCR C instruction and save a byte of object code. Why are we allowed to do this?

## Replacing A Chain With Data

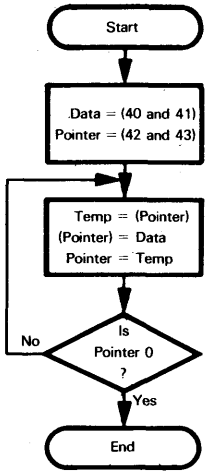
**Purpose:** Replace each entry in a chain of addresses with data. The data is in memory locations 40 and 41 (MSBs in 41). The address of the start of the chain is in memory locations 42 and 43 (MSBs in 43). Each entry in the chain is two bytes long and points to the address of the next two-byte element in the chain. The last element in the chain contains zero to indicate that there is no next element.

This kind of data structure (the chain) is used to allow forward referencing, i.e., referencing symbols before they are defined. The most common use is in assemblers and compilers. In this usage, when a symbol that has not yet been defined (by defined we mean established by an EQU or used as a label, etc.) is referenced, an entry is made to a chain of all references to that symbol. When the symbol is finally defined, all references to that symbol are located and then satisfied by processing the chain.

**Sample Problem:**

```
(40) = 66 }
(41) = 00 } data

(42) = 46 }
(43) = 00 } address of first element in list

(46) = 4D }
(47) = 00 } address of second element in list

(4D) = 00 }
(4E) = 00 } end of list.

Result = (46) = 66
         (47) = 00
         (4D) = 66
         (4E) = 00
```

**Flowchart:**




**Source Program:**

```
        LHLD    40H     ;GET DATA AND SAVE
        MOV     B,H
        MOV     C,L
        LHLD    42H     ;POINT TO START OF LIST
CHAIN:  MOV     E,M     ;GET ADDRESS OF NEXT ELEMENT
        MOV     M,C
        INX     H
        MOV     D,M
        MOV     M,B     ;REPLACE POINTER WITH DATA
        XCHG            ;NEW POINTER
        MOV     A,H     ;IS NEW POINTER ZERO?
        ORA     L       ;IF SO, NO MORE ELEMENTS
        JNZ     CHAIN   ;IF NOT, CONTINUE TRACING
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LHLD | 40H | 2A |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MOV | B,H | 44 |
| 04 | | MOV | C,L | 4D |
| 05 | | LHLD | 42H | 2A |
| 06 | | | | 42 |
| 07 | | | | 00 |
| 08 | CHAIN: | MOV | E,M | 5E |
| 09 | | MOV | M,C | 71 |
| 0A | | INX | H | 23 |
| 0B | | MOV | D,M | 56 |
| 0C | | MOV | M,B | 70 |
| 0D | | XCHG | | EB |
| 0E | | MOV | A,H | 7C |
| 0F | | ORA | L | B5 |
| 10 | | JNZ | CHAIN | C2 |
| 11 | | | | 08 |
| 12 | | | | 00 |
| 13 | HERE: | JMP | HERE | C3 |
| 14 | | | | 13 |
| 15 | | | | 00 |

The 8080 processor has some 16-bit instructions but not a full set. There is no implied memory addressing with 16-bit data nor is there a way of checking if a 16-bit number is zero. We therefore must use the 8-bit MOV instruction to fetch the new pointer and fill the old one with data. We also must use the logical ORing of the two 8-bit sections to determine if the 16-bit pointer is zero (note that adding the two 8-bit sections would not work — why?)

The XCHG instruction which exchanges the contents of register pair H and L with D and E is convenient as a 16-bit register transfer instruction. XCHG replaces a whole series of MOV instructions since it exchanges two 16-bit quantities. Note that you cannot transfer the pointer immediately to H and L (what happens if you replace MOV E,M with MOV L,M?)

Chaining can handle lists which are not in sequential memory locations. Each element must somehow contain the address of the next element. Such lists can allow a user to change variables, fill in definitions in a program, or create a linked data base.

## 8-Bit Sort

**Purpose:** Sort an array of unsigned binary numbers into decreasing order. The length of the array is in memory location 40 and the array itself begins in memory location 41.

**Sample Problem:**

(40) = 06
(41) = 2A
(42) = B5
(43) = 60
(44) = 3F
(45) = D1
(46) = 19

Result = (41) = D1
(42) = B5
(43) = 60
(44) = 3F
(45) = 2A
(46) = 19

A simple sorting technique works as follows:

**SIMPLE SORTING ALGORITHM**

STEP 1 - Clear a flag INTER.

STEP 2 - Successively examine each pair of numbers in the array. If any are out of order, exchange them and set flag INTER.

STEP 3 - If INTER ≠ 0, return to STEP 1.

INTER will be set to 1 if any pair of numbers is out of order. Therefore, if INTER = 0, the array is in proper order.

The technique operates as follows in a simple case. Let us assume that we want to sort an array into ascending order; the array has four elements — 08, 15, 03, 12.

1st iteration:

STEP 1 - INTER = 0

STEP 2 - Final order of the array is:

15
08
12
03

since the first pair (08, 15) is exchanged and so is the third pair (03, 12). INTER = 1.

2nd iteration:

STEP 1 - INTER = 0

STEP 2 - Final order of the array is:

15
12
08
03

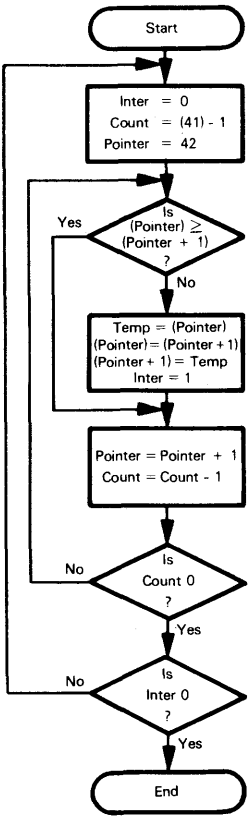
since the second pair (08, 12) is exchanged. INTER = 1.

3rd iteration:

STEP 1 - INTER = 0

STEP 2 - The elements are already in order so no exchanges are necessary and INTER remains zero.

**Flowchart:**


Start
Inter = 0
Count = (41) - 1
Pointer = 42
Is
(Pointer) ≥
(Pointer + 1)
?
Yes
No
Temp = (Pointer)
(Pointer) = (Pointer + 1)
(Pointer + 1) = Temp
Inter = 1
Pointer = Pointer + 1
Count = Count - 1
Is
Count 0
?
No
Yes
Is
Inter 0
?
No
Yes
End

**Source Program:**

```
SORT:   MVI     B,0         ;INTERCHANGE FLAG = 0
        LXI     H,41H       ;COUNT = LENGTH OF ARRAY
        MOV     C,M
        DCR     C           ;NUMBER OF PAIRS = COUNT -1
        INX     H           ;POINT TO START OF ARRAY
PASS1:  MOV     A,M         ;GET Kth ELEMENT
        INX     H
        CMP     M           ;COMPARE TO (K+1)th ELEMENT
        JNC     CNT         ;NO INTERCHANGE IF Kth ≥ (K+1)th
        MOV     D,M         ;INTERCHANGE IF OUT OF ORDER
        MOV     M,A
        DCX     H
        MOV     M,D
        INX     H
        MVI     B,1         ;INTERCHANGE FLAG = 1
CNT:    DCR     C           ;COUNT DOWN
        JNZ     PASS1
        DCR     B           ;IS INTERCHANGE FLAG 1?
        JZ      SORT        ;YES, DO ANOTHER PASS
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | SORT: | MVI | B,0 | 06 |
| 01 | | | | 00 |
| 02 | | LXI | H,41H | 21 |
| 03 | | | | 41 |
| 04 | | | | 00 |
| 05 | | MOV | C,M | 4E |
| 06 | | DCR | C | 0D |
| 07 | | INX | H | 23 |
| 08 | PASS1: | MOV | A,M | 7E |
| 09 | | INX | H | 23 |
| 0A | | CMP | M | BE |
| 0B | | JNC | CNT | D2 |
| 0C | | | | 15 |
| 0D | | | | 00 |
| 0E | | MOV | D,M | 56 |
| 0F | | MOV | M,A | 77 |
| 10 | | DCX | H | 2B |
| 11 | | MOV | M,D | 72 |
| 12 | | INX | H | 23 |
| 13 | | MVI | B,1 | 06 |
| 14 | | | | 01 |
| 15 | CNT: | DCR | C | 0D |
| 16 | | JNZ | PASS1 | C2 |
| 17 | | | | 08 |
| 18 | | | | 00 |
| 19 | | DCR | B | 05 |
| 1A | | JZ | SORT | CA |
| 1B | | | | 00 |
| 1C | | | | 00 |
| 1D | HERE: | JMP | HERE | C3 |
| 1E | | | | 1D |
| 1F | | | | 00 |

The case where two elements in the array are equal is very important here. The program should not perform an interchange in that case since that interchange would occur in each pass. The result would be that each pass would set the interchange flag, thus producing an endless loop.

The 8080 Conditional Branch instructions can be limiting, and are particularly limiting in this program. Following an instruction like CMP M, we have only JC, jump if (M) > (A), and JNC, jump if (M) $\leq$ (A). We do not have Jump instructions in the case where the equality condition is reversed, i.e., (M) $\geq$ (A) and (M) < (A). Therefore, we must be careful of the order of operations.

Implied memory addressing through Registers H and L is awkward here since the program uses pairs of elements rather than single elements. Note that if the 8080 had indexing this program would be much simpler, since the two elements could be referred to with the same base address but different indexes.

Before starting each sorting pass, we must be careful to re-initialize the counter, pointer, and interchange flag.

**OTHER SORTING METHODS**

There are many sorting algorithms which vary widely in efficiency. Knuth describes some in the book mentioned earlier ("The Art Of Computer Programming, Volume III: Sorting and Searching"). Kernighan and Plauger describe some algorithms and compare their efficiency on pages 106-111 of their book "The Elements of Programming Style", McGraw-Hill, New York, 1974.

## Using A Jump Table With A Key

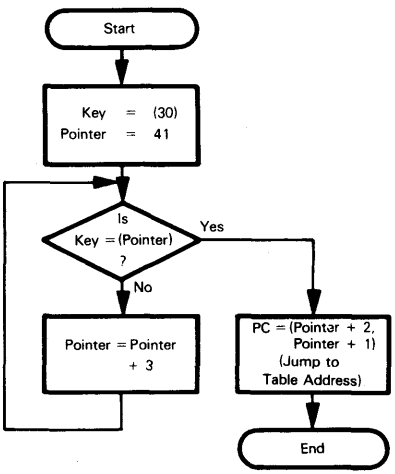
**Purpose:** Use the contents of memory location 30 as the key to a jump table starting in location 41. Each entry in the jump table contains an 8-bit key value followed by a 16-bit address (MSBs in second word) to which the program should transfer control if the key is equal to that key value.

**Sample Problem:**

(30) = 38
(41) = 32
(42) = 55
(43) = 00
(44) = 35
(45) = 61
(46) = 00
(47) = 38
(48) = 65
(49) = 00

Result = (PC) = 0065 since that address corresponds to key value 38.

**Flowchart:**



**Source Program:**

```
        LDA     30H     ;GET KEY
        LXI     H,41H   ;POINT TO START OF JUMP TABLE
SRKEY:  CMP     M       ;IS KEY = TABLE KEY?
        INX     H
        JZ      FOUND   ;YES, JUMP TO ADDRESS IN TABLE
        INX     H       ;NO, GO TO NEXT ENTRY
        INX     H
        JMP     SRKEY
FOUND:  MOV     E,M     ;GET JUMP ADDRESS FROM TABLE
        INX     H
        MOV     D,M
        XCHG
        PCHL            ;AND JUMP TO IT BY MOVING IT TO PC
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LDA | 30H | 3A |
| 01 | | | | 30 |
| 02 | | | | 00 |
| 03 | | LXI | H,41H | 21 |
| 04 | | | | 41 |
| 05 | | | | 00 |
| 06 | SRKEY: | CMP | M | BE |
| 07 | | INX | H | 23 |
| 08 | | JZ | FOUND | CA |
| 09 | | | | 10 |
| 0A | | | | 00 |
| 0B | | INX | H | 23 |
| 0C | | INX | H | 23 |
| 0D | | JMP | SRKEY | C3 |
| 0E | | | | 06 |
| 0F | | | | 00 |
| 10 | FOUND: | MOV | E,M | 5E |
| 11 | | INX | H | 23 |
| 12 | | MOV | D,M | 56 |
| 13 | | XCHG | | EB |
| 14 | | PCHL | | E9 |

If a key comparison fails, we must skip over the key and the associated address before making the next comparison. The three INX H instructions do this.

The instruction PCHL, which transfers H and L to the Program Counter, is very handy in jump tables and monitor programs. Note that PCHL is a Jump instruction since it places a new value in the Program Counter. It is, in fact, the only 8080 instruction that allows us to place a variable address directly into the Program Counter. All the Jump and Call instructions use fixed addresses.

No ending operation is necessary since PCHL transfers control to the address in the jump table.

Jump tables are very useful in situations where one of several routines must be selected. Such situations arise in decoding commands, selecting test programs, choosing alternate methods, or selecting an I/O configuration.

The jump table replaces a whole series of conditional Jump operations. The program which searches the jump table could be used to search several different tables merely by changing the key and starting address.

Here again, handling 16-bit or longer data is awkward. The 8080 has no single instruction that will increment the address in H and L by three, nor does it have a single instruction that will fetch the 16-bit jump address from the table. Each of these tasks requires several instructions.

How could you restructure the initial conditions to eliminate the extra Jump instruction?

## PROBLEMS

### 1) Remove Entry From List

**Purpose:** Remove the contents of memory location 30 from a list if it is present. The length of the list is in memory location 41 and the list itself begins in memory location 42. Move the entries below the one removed up one position and reduce the length of the list by 1.

**Sample Problems:**

a.
(30) = 6B
(41) = 04
(42) = 37
(43) = 61
(44) = 28
(45) = 1D
Result = No change since the entry is not in the list.

b.
(30) = 6B
(41) = 04
(42) = 37
(43) = 6B
(44) = 28
(45) = 1D
Result = (41) = 03
(43) = 28
(44) = 1D

The entry is removed from the list and the ones below it are moved up one position. The length of the list is reduced by 1.

## 2) Add Entry To Ordered List

**Purpose:** Place the contents of memory location 30 in an ordered list if it is not already there. The length of the list is in memory location 41; the list itself begins in memory location 42 and consists of unsigned binary numbers in increasing order. Place the new entry in the correct position in the list, adjust the elements below it down, and increase the length of the list by 1.

**Sample Problems:**

a.

(30) = 6B
(41) = 04
(42) = 37
(43) = 55
(44) = 7D
(45) = A1

Result = (41) = 05
(44) = 6B
(45) = 7D
(46) = A1

b.

(30) = 6B
(41) = 04
(42) = 37
(43) = 55
(44) = 6B
(45) = A1

Result = Unchanged since the entry is already in the list.

## 3) Add Element To Chained List

**Purpose:** Add the address in memory location 40 and 41 (MSBs in 41) to a chained list. The address of the old starting address of the list is in memory location 42 and 43 (MSBs in 43). Each entry in the chained list contains either the address of the next element in the list or zero if there is no next element; the entries are all 16 bits with the most significant bits in the second word of the entry. The new entry goes at the head of the list; its address will be in memory location 42 and 43 and it will contain the address that was previously in those locations.

**Sample Problem:**

(40) = 46 } new entry
(41) = 00 }

(42) = 4D } pointer to old head of list
(43) = 00 }

Result = (42) = 46 } pointer to new head of list
(43) = 00 }

(46) = 4D } entry points to old head of list
(47) = 00 }

## 4) 16-Bit Sort

**Purpose:** Sort an array of unsigned 16-bit binary numbers into increasing order. The length of the array is in memory location 40 and the array itself begins in memory location 41. Each 16-bit number is stored with the least significant bits in the first word.

**Sample Problem:**

(40) = 03
(41) = 2A
(42) = B5
(43) = 60
(44) = 3F
(45) = D1
(46) = 19

Result = (41) = D1
(42) = 19
(43) = 60
(44) = 3F
(45) = 2A
(46) = B5

The numbers are B52A, 3F60, and 19D1.

## 5) Using An Ordered Jump Table

**Purpose:** Use the contents of memory location 30 as an index to a jump table starting in location 41. Each entry in the jump table contains a 16-bit address with the MSBs in the second word to which the program should transfer control if the index has the appropriate value, i.e., if the index is 6, the program forces a jump to address entry #6 in the table.

**Sample Problem:**

(30) = 02
(41) = 00
(42) = 50
(43) = 00
(44) = 56
(45) = 00
(46) = 60

Result = (PC) = 0060 since that is entry #2 in the jump table.

# Chapter 10
# SUBROUTINES

None of the examples that we have shown so far is typically a program all by itself. Most real programs perform a series of tasks, many of which may be the same or may be common to several different programs. We need a way to formulate these tasks once and make that formulation conveniently available both in different parts of the current program and in other programs.

**SUBROUTINE LIBRARY**

The answer is to formulate the task as a "subroutine". The resulting sequence of instructions can be written once, tested, and then used repeatedly. It can form part of a "subroutine library" which will provide documented solutions to common problems.

**SUBROUTINE INSTRUCTIONS**

Most microprocessors have special instructions for transferring control to subroutines and restoring control to the main program. We often refer to the special instruction that transfers control to a subroutine as Call, Jump to Subroutine, Jump and Mark Place, or Jump and Link. The special instruction that restores control to the main program is usually called Return. On the 8080 microprocessor, the Call instruction saves the old value of the Program Counter in the RAM Stack before placing the starting address of the subroutine in the Program Counter; the Return instruction gets the old value from the Stack and puts it back in the Program Counter. The effect is to transfer program control, first to the subroutine and then back to the main program. Clearly the subroutine may itself transfer control to a subroutine and so on.

In order to be really useful, a subroutine must be reasonably general. A routine that can only perform a specialized task such as looking for a particular letter in an input string of fixed length will not be very useful. If, on the other hand, the subroutine can look for any letters in strings of any length, it will be far more helpful. We call the data or addresses which the subroutine permits to be variables "parameters". An important part of writing subroutines is deciding which variables should be parameters.

**PASSING PARAMETERS**

One problem is transferring the parameters to the subroutine; this process is called "passing" parameters. The simplest method is for the main program to place the parameters in registers. Then the subroutine can simply assume that the parameters are there. Of course, this technique is limited by the number of registers that are available. The parameters may, however, be addresses as well as data. For example, a sorting routine could begin with the starting address of an array in Registers H and L.

Other methods are necessary when there are more parameters. One possibility is to use the Stack. The main program can place the parameters in the Stack and the subroutine can retrieve them. The advantages of this method are that the Stack is essentially unlimited in size and that data in the Stack is not lost even if the Stack is used again. The disadvantages are that few 8080 instructions use the Stack, and the Call instruction also stores the return address in the Stack. Another method is to use an area of memory for parameters. The main program can place the address of the area in Registers H and L and the subroutine can then retrieve the data as needed. However, this procedure is awkward if the parameters themselves are addresses.

**RELOCATION**

Sometimes a subroutine must have special features. A subroutine is "relocatable" if it can be placed anywhere in memory. You can use such a subroutine easily regardless of the placement of other programs or the arrangement of the memory. A strictly relocatable program can use no absolute addresses; all addresses must be relative to the start of the program. The program requires a "relocating loader" to place it in memory; the loader will start the program after other programs and will add the starting address or "relocation constant" to all addresses in the program.

**RE-ENTRANT SUBROUTINE**

A subroutine is "re-entrant" if it can be interrupted and re-entered by the interrupting program. Re-entrancy is important for standard subroutines in an interrupt-based system. Otherwise the interrupt service routines cannot use the standard subroutines without producing errors. Microprocessor subroutines are easy to make re-entrant since the Call instruction uses the Stack and that procedure is automatically re-entrant. The only remaining requirement is that the subroutine must use the registers and Stack rather than fixed memory locations for temporary storage. This is a bit awkward but usually can be done if necessary.

A subroutine is "recursive" if it calls itself. Such a subroutine clearly must also be re-entrant. However, recursive subroutines are uncommon in microprocessor applications.

Most programs consist of a main program and several subroutines. This is advantageous because you can use proven routines and debug and test the other subroutines separately. You must, however, be careful to use the subroutines properly and remember their exact effects.

## SUBROUTINE DOCUMENTATION

**DOCUMENTING SUBROUTINES**

Subroutine listings must provide enough information so that other users can utilize the subroutine without having to examine its internal structure. Among the necessary specifications are:

1) A description of the purpose of the subroutine.
2) A list of parameters.
3) Registers and memory locations used.
4) A sample case.

If these guidelines are followed, the subroutine will be as easy to use as possible.

It is important to note that the following examples all reserve an area of memory for the RAM stack. If the monitor in your microcomputer establishes such an area, you may use it instead. If you wish to try establishing your own stack area, remember to save and restore the monitor's Stack Pointer in order to produce a proper return at the end of your main program.

To save the monitor Stack Pointer, use the routine:

```
LXI     H,0         ;GET MONITOR STACK POINTER
DAD     SP
SHLD    STEMP       ;AND SAVE IT
```

To restore the monitor Stack Pointer, use the routine:

```
LHLD    STEMP       ;RESTORE MONITOR STACK POINTER
SPHL
```

We have used 80 hex as the starting point for the Stack. If necessary, you should consistently replace that address with one suitable for your microcomputer.

## EXAMPLES

## Hex To ASCII

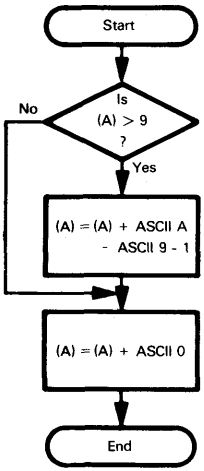
**Purpose:** Convert the contents of the Accumulator to an ASCII character. Place the result in the Accumulator.

**Sample Problems:**

a. (A) = 0C
Result = (A) = 43 'C'

b. (A) = 06
Result = (A) = 36 '6'

**Flowchart:**



**Source Program:**

The calling program starts the Stack at memory location 80, gets the binary data from memory location 40, calls the conversion subroutine, and stores the result in memory location 41.

```
        ORG     0
        LXI     SP,80H      ;START STACK AT MEMORY LOCATION 80
        LDA     40H         ;GET DATA
        CALL    ASDEC       ;CONVERT TO ASCII
        STA     41H         ;STORE RESULT
HERE:   JMP     HERE
```

The subroutine converts a hexadecimal digit to ASCII.

```
        ORG     20H
ASDEC:  CPI     10          ;IS DATA 10 OR MORE?
        JNC     ASCZ
        ADI     'A'-'9'-1   ;YES, ADD OFFSET FOR LETTERS
ASCZ:   ADI     '0'         ;ADD OFFSET FOR ASCII
        RET
```

Subroutine documentation:

```
;
;SUBROUTINE ASDEC
;
;PURPOSE: ASDEC CONVERTS A HEXADECIMAL
;  DIGIT IN THE ACCUMULATOR TO AN
;  ASCII CHARACTER IN THE ACCUMULATOR
;
;INITIAL CONDITIONS: HEX DIGIT IN A
;
;FINAL CONDITIONS: ASCII CHARACTER IN A
;
;REGISTERS USED: A
;
;SAMPLE CASE
;  STARTING CONDITION: 6 IN ACCUMULATOR
;  FINAL CONDITION: ASCII 6 (HEX 36)
;    IN ACCUMULATOR
;
```

**Object Program:**

Calling program:

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | SP,80H | 31 |
| 01 | | | | 80 |
| 02 | | | | 00 |
| 03 | | LDA | 40H | 3A |
| 04 | | | | 40 |
| 05 | | | | 00 |
| 06 | | CALL | ASDEC | CD |
| 07 | | | | 20 |
| 08 | | | | 00 |
| 09 | | STA | 41H | 32 |
| 0A | | | | 40 |
| 0B | | | | 00 |
| 0C | HERE: | JMP | HERE | C3 |
| 0D | | | | 0C |
| 0E | | | | 00 |

Subroutine:

| | | | | |
|---|---|---|---|---|
| 20 | ASDEC: | CPI | 10 | FE |
| 21 | | | | 0A |
| 22 | | JNC | ASCZ | D2 |
| 23 | | | | 27 |
| 24 | | | | 00 |
| 25 | | ADI | 'A'-'9'-1 | C6 |
| 26 | | | | 07 |
| 27 | ASCZ: | ADI | '0' | C6 |
| 28 | | | | 30 |
| 29 | | RET | | C9 |

The instruction LXI SP,80H starts the Stack at memory location 80. Remember that the Stack grows downward (to lower addresses). We usually place the Stack at the high end of RAM (i.e., the largest address) so that it will not interfere with other temporary storage.

The Call instruction places the subroutine starting address (0020 hex) in the Program Counter and saves the old Program Counter (0009 hex) in the Stack. The procedure is:

STEP 1 - Decrement Stack Pointer, save MSBs of old Program Counter in Stack.

STEP 2 - Decrement Stack Pointer, save LSBs of old Program Counter in Stack.

The result in this case is:

(7F) = 00
(7E) = 09
(SP) = 7E

The value which is saved is the value of the Program Counter after the processor has fetched the entire Call instruction from memory. Note that the address ends up stored just like other 8080 addresses with the least significant bits in the lower address.

The Return instruction loads the Program Counter with the contents of the bottom two memory locations in the Stack into the Program Counter. The procedure is:

STEP 1 - Move 8 bits from Stack to LSBs of Program Counter. Increment Stack Pointer.

STEP 2 - Move 8 bits from Stack to MSBs of Program Counter. Increment Stack Pointer.

The result in this case is:

$$\begin{aligned} [PC] &= (7F)(7E) \\ &= 0009 \\ [SP] &= 80 \end{aligned}$$

This subroutine has a single parameter and produces a single result. The Accumulator is the obvious place to put both.

The calling program involves three steps: placing the data in the Accumulator, calling the subroutine, and storing the result. In addition, the initialization must assign the Stack to an appropriate area of memory.

The subroutine is re-entrant since it uses no memory locations. The subroutine can be relocated by using a different ORG statement and then reassembling the code.

If you plan to use the Stack for parameters, remember that the Call instruction places the return address in the Stack. You can execute INX SP twice to get past the return address but you must also remember to adjust the Stack Pointer properly before returning. You can also move the Stack Pointer to Registers H and L with the sequence:

```
LXI     H,0
DAD     SP          ;STACK POINTER TO ADDRESS REGISTER
```

Now you can use implied memory addressing with H and L to access data in the Stack.

## Length Of A String Of Characters

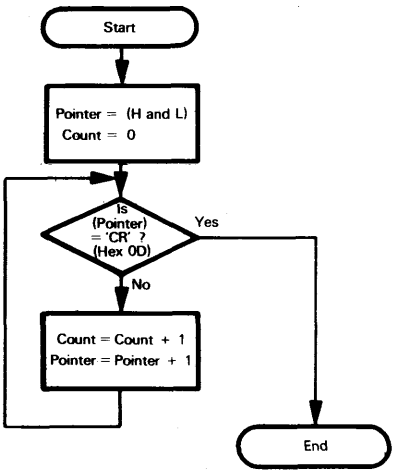
**Purpose:** Determine the length of a string of ASCII characters. The starting address of the string is in Registers H and L. The end of the string is marked by a carriage return character ('CR', hex 0D). Place the length of the string (excluding the carriage return) in the Accumulator.

**Sample Problems:**

a.
(H AND L) = 43
(43) = 0D
Result = (A) = 00

b.
(H AND L) = 43
(43) = 52
(44) = 41
(45) = 54
(46) = 48
(47) = 45
(48) = 52
(49) = 0D
Result = (A) = 06

**Flowchart:**



**Source Program:**

The calling program starts the Stack at memory location 80, gets the starting address from memory locations 40 and 41, calls the string length subroutine, and stores the result in memory location 42.

```
        ORG     0
        LXI     SP,80H      ;START STACK AT LOCATION 80
        LHLD    40H         ;GET STARTING ADDRESS OF STRING
        CALL    STLEN       ;DETERMINE STRING LENGTH
        STA     42H         ;STORE STRING LENGTH
HERE:   JMP     HERE
```

The subroutine determines the length of a string of ASCII characters and places the length in the Accumulator.

```
        ORG     20H
STLEN:  MVI     B,0         ;LENGTH = 0
        MVI     A,0DH       ;GET 'CR' FOR COMPARISON
CHKCR:  CMP     M           ;IS CHARACTER 'CR'?
        JZ      DONE        ;YES, END OF STRING
        INR     B           ;NO, ADD 1 TO LENGTH
        INX     H
        JMP     CHKCR
DONE:   MOV     A,B
        RET
```

Subroutine documentation:

```
;
;SUBROUTINE STLEN
;
;PURPOSE: STLEN DETERMINES THE LENGTH
;  OF A STRING (NUMBER OF CHARACTERS
;  BEFORE A CARRIAGE RETURN)
;
;INITIAL CONDITIONS: STARTING ADDRESS
;  OF STRING IN H AND L
;
;FINAL CONDITIONS: NUMBER OF CHARACTERS IN A
;
;REGISTERS USED: A, B, H, L
;
;SAMPLE CASE:
;  STARTING CONDITION: (H AND L) = 43
;     (43) = 35, (44) = 44, (45) = 0D
;  FINAL CONDITION: (A) = 02
;
```

**Object Program:**

Calling program:

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | SP,80H | 31 |
| 01 | | | | 80 |
| 02 | | | | 00 |
| 03 | | LHLD | 40H | 2A |
| 04 | | | | 40 |
| 05 | | | | 00 |
| 06 | | CALL | STLEN | CD |
| 07 | | | | 20 |
| 08 | | | | 00 |
| 09 | | STA | 42H | 32 |
| 0A | | | | 42 |
| 0B | | | | 00 |
| 0C | HERE: | JMP | HERE | C3 |
| 0D | | | | 0C |
| 0E | | | | 00 |

Subroutine:

| | | | | |
|---|---|---|---|---|
| 20 | STLEN: | MVI | B,0 | 06 |
| 21 | | | | 00 |
| 22 | | MVI | A,0DH | 3E |
| 23 | | | | 0D |
| 24 | CHKCR: | CMP | M | BE |
| 25 | | JZ | DONE | CA |
| 26 | | | | 2D |
| 27 | | | | 00 |
| 28 | | INR | B | 04 |
| 29 | | INX | H | 23 |
| 2A | | JMP | CHKCR | C3 |
| 2B | | | | 24 |
| 2C | | | | 00 |
| 2D | DONE: | MOV | A,B | 78 |
| 2E | | RET | | C9 |

The calling program involves four steps: initializing the Stack Pointer, placing the starting address of the string in Registers H and L, calling the subroutine, and storing the result.

The subroutine is re-entrant since it does not change any memory locations.

The subroutine changes Register B and the address in Registers H and L as well as the Accumulator. The programmer must be aware that data stored in Register B and the address loaded into H and L will be lost; the registers used are an absolutely essential part of the subroutine documentation.

An alternative to destroying register contents in the subroutine is to save them in the Stack and then restore them before returning. This approach makes life easier for the user calling the routine, but costs extra time and memory (in the Stack) in the routine.

This subroutine has a single parameter which is an address. The best way to pass this parameter is through a register pair, and since the H and L register pair is certainly the most flexible as far as addressing options are concerned, it is the obvious choice.

The subroutine contains an unconditional Jump instruction, JMP CHKCR. By altering the initial conditions prior to entering the subroutine's loop, can you eliminate this Jump?

## Add Even Parity To ASCII Characters

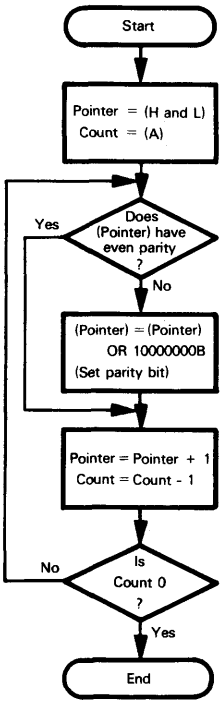
**Purpose:** Add even parity to a string of 7-bit ASCII characters. The length of the string is in the Accumulator and the starting address of the string is in Registers H and L. Place even parity in the most significant bit of each character, i.e., set the most significant bit if that makes the total number of 1 bits in the word even.

**Sample Problem:**

(A) = 6
(H AND L) = 40
(40) = 31
(41) = 32
(42) = 33
(43) = 34
(44) = 35
(45) = 36

Result = (40) = B1
(41) = B2
(42) = 33
(43) = B4
(44) = 35
(45) = 36

**Flowchart:**


Start
Pointer = (H and L)
Count = (A)
Does (Pointer) have even parity ?
Yes
No
(Pointer) = (Pointer) OR 10000000B (Set parity bit)
Pointer = Pointer + 1
Count = Count - 1
Is Count 0 ?
No
Yes
End

**Source Program:**

The calling program starts the Stack at memory location 80, sets the starting address to 40, gets the string length from memory location 30, and calls the even parity subroutine.

```
        ORG     0
        LXI     SP,80H          ;START STACK AT LOCATION 80
        LXI     H,40H           ;GET STARTING ADDRESS OF STRING
        LDA     30H             ;GET STRING LENGTH
        CALL    EPAR            ;ADD EVEN PARITY TO STRING
HERE:   JMP     HERE
```

The subroutine adds even parity to a string of ASCII characters.

```
        ORG     20H
EPAR:   MOV     B,A
        MVI     C,10000000B     ;GET PARITY BIT OF 1
SETPR:  MOV     A,M             ;GET A CHARACTER
        ORA     C               ;PARITY BIT = 1
        JPO     CHCNT           ;DON'T SAVE IF PARITY NOW ODD
        MOV     M,A             ;MAKE PARITY EVEN
CHCNT:  INX     H
        DCR     B
        JNZ     SETPR
        RET
```

Subroutine documentation:

```
;
;SUBROUTINE EPAR
;
;PURPOSE: EPAR ADDS EVEN PARITY
;   TO A STRING OF 7-BIT ASCII
;   CHARACTERS
;
;INITIAL CONDITIONS: STARTING ADDRESS
;   OF STRING IN H AND L, LENGTH
;   OF STRING IN A
;
;FINAL CONDITIONS: EVEN PARITY IN
;      MSB OF EACH CHARACTER
;
;REGISTERS USED: A, B, C, H, L
;
;SAMPLE CASE:
;   STARTING CONDITION: (H AND L) = 40
;      (A) = 2, (40) = 32, (41) = 33
;   FINAL CONDITION: (40) = B2,
;      (41) = 33
;
```

**Object Program:**

Calling program:

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | SP,80H | 31 |
| 01 | | | | 80 |
| 02 | | | | 00 |
| 03 | | LXI | H,40H | 21 |
| 04 | | | | 40 |
| 05 | | | | 00 |
| 06 | | LDA | 30H | 3A |
| 07 | | | | 30 |
| 08 | | | | 00 |
| 09 | | CALL | EPAR | CD |
| 0A | | | | 20 |
| 0B | | | | 00 |
| 0C | HERE: | JMP | HERE | C3 |
| 0D | | | | 0C |
| 0E | | | | 00 |

Subroutine:

| | | | | |
|---|---|---|---|---|
| 20 | EPAR: | MOV | B,A | 47 |
| 21 | | MVI | C,10000000B | 0E |
| 22 | | | | 80 |
| 23 | SETPR: | MOV | A,M | 7E |
| 24 | | ORA | C | B1 |
| 25 | | JPO | CHCNT | E2 |
| 26 | | | | 29 |
| 27 | | | | 00 |
| 28 | | MOV | M,A | 77 |
| 29 | CHCNT: | INX | H | 23 |
| 2A | | DCR | B | 05 |
| 2B | | JNZ | SETPR | C2 |
| 2C | | | | 23 |
| 2D | | | | 00 |
| 2E | | RET | | C9 |

The calling program must place the starting address of the string in Registers H and L and the length of the string in A before transferring control to the subroutine.

The subroutine changes the values in Registers A, H, and L and uses Registers B and C for temporary storage. It is re-entrant since it does not use any fixed memory locations for temporary storage.

This subroutine has two parameters, an address and a number. Registers H and L are used to pass the address and A to pass the number. No explicit results are returned since the subroutine only affects the MSB of each character in the string.

## Pattern Match

**Purpose:** Compare two strings of ASCII characters to see if they are the same. The length of the strings is in the Accumulator. The starting address of one string is in Registers H and L; the starting address of the other string is in Registers D and E. If the two strings match, clear the Accumulator; otherwise, set the Accumulator to FF hex.

**Sample Problems:**

a.

(A) = 03
(D AND E) = 50
(H AND L) = 60

(50) = 43 C
(51) = 41 A
(52) = 54 T

(60) = 43 C
(61) = 41 A
(62) = 54 T

Result = (A) = 0

b.

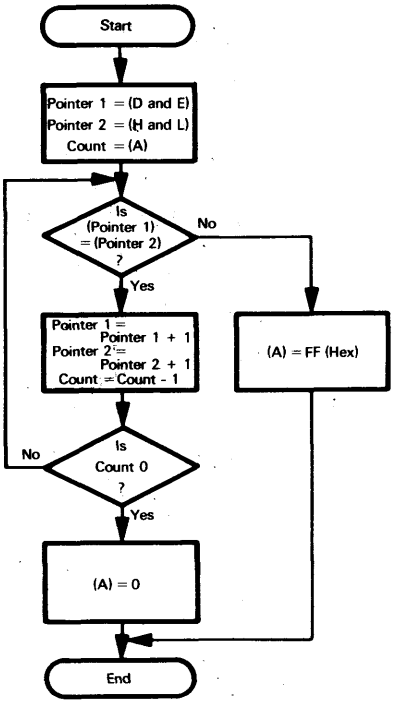
(A) = 03
(D AND E) = 50
(H AND L) = 60

(50) = 52 R
(51) = 41 A
(52) = 54 T

(60) = 43 C
(61) = 41 A
(62) = 54 T

Result = (A) = FF (hex)

**Flowchart:**


Start
Pointer 1 = (D and E)
Pointer 2 = (H and L)
Count = (A)
Is (Pointer 1) = (Pointer 2) ?
No
Yes
Pointer 1 = Pointer 1 + 1
Pointer 2 = Pointer 2 + 1
Count = Count - 1
(A) = FF (Hex)
Is Count 0 ?
No
Yes
(A) = 0
End

**Source Program:**

The calling program starts the Stack at memory location 80, sets the starting addresses of the strings to 50 and 60 respectively, gets the string length from memory location 40, calls the pattern match subroutine, and places the result in memory location 41.

```
        ORG     0
        LXI     SP,80H    ;START STACK AT LOCATION 80
        LXI     D,60H     ;GET STARTING ADDRESS OF STRING 1
        LXI     H,50H     ;GET STARTING ADDRESS OF STRING 2
        LDA     40H       ;GET STRING LENGTH
        CALL    PMTCH     ;CHECK FOR MATCH
        STA     41H       ;SAVE MATCH INDICATOR
HERE:   JMP     HERE
```

The subroutine determines if the two strings are the same.

```
        ORG     20H
PMTCH:  MOV     B,A
CHCAR:  LDAX    D         ;GET CHARACTER FROM STRING 2
        CMP     M         ;IS THERE A MATCH WITH STRING 1?
        JNZ     NOMCH     ;NO, DONE — STRINGS NOT EQUAL
        INX     D         ;MOVE POINTERS
        INX     H
        DCR     B         ;COUNT CHARACTERS
        JNZ     CHCAR
        SUB     A         ;COMPLETE MATCH, (A) = 0
        RET
NOMCH:  MVI     A,0FFH    ;NO MATCH, (A) = FF
        RET
```

Subroutine documentation:

```
;
;SUBROUTINE PMTCH
;
;PURPOSE: PMTCH DETERMINES IF
;   TWO STRINGS ARE EQUIVALENT
;
;INITIAL CONDITIONS: STARTING ADDRESSES
;   OF STRINGS IN D AND E, H AND L;
;   LENGTH OF STRINGS IN ACCUMULATOR
;
;FINAL CONDITIONS: 0 IN A IF
;   STRINGS MATCH, FF IN A OTHERWISE
;
;REGISTERS USED: A, B, D, E, H, L
;
;SAMPLE CASE:
;   STARTING CONDITIONS: (H AND L) =
;      50, (D AND E) = 60, (A) = 2
;      (50) = 36, (51) = 39
;      (60) = 36, (61) = 39
;   FINAL CONDITION: (A) = 0 SINCE THE STRINGS MATCH
```

**Object Program:**

Calling program:

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | SP,80H | 31 |
| 01 | | | | 80 |
| 02 | | | | 00 |
| 03 | | LXI | D,60H | 11 |
| 04 | | | | 60 |
| 05 | | | | 00 |
| 06 | | LXI | H,50H | 21 |
| 07 | | | | 50 |
| 08 | | | | 00 |
| 09 | | LDA | 40H | 3A |
| 0A | | | | 40 |
| 0B | | | | 00 |
| 0C | | CALL | PMTCH | CD |
| 0D | | | | 20 |
| 0E | | | | 00 |
| 0F | | STA | 41H | 32 |
| 10 | | | | 41 |
| 11 | | | | 00 |
| 12 | HERE: | JMP | HERE | C3 |
| 13 | | | | 12 |
| 14 | | | | 00 |

Subroutine:

| | | | | |
|---|---|---|---|---|
| 20 | PMTCH: | MOV | B,A | 47 |
| 21 | CHCAR: | LDAX | D | 1A |
| 22 | | CMP | M | BE |
| 23 | | JNZ | NOMCH | C2 |
| 24 | | | | 2E |
| 25 | | | | 00 |
| 26 | | INX | D | 13 |
| 27 | | INX | H | 23 |
| 28 | | DCR | B | 05 |
| 29 | | JNZ | CHCR | C2 |
| 2A | | | | 21 |
| 2B | | | | 00 |
| 2C | | SUB | A | 97 |
| 2D | | RET | | C9 |
| 2E | NOMCH: | MVI | A,0FFH | 3E |
| 2F | | | | FF |
| 30 | | RET | | C9 |

This subroutine, like the preceding ones, changes all of the flags. You should generally assume that a subroutine call changes the flags unless it is specifically stated otherwise. If the main program needs the old flag values, it must save them in the Stack prior to calling the subroutine. (This is accomplished by using the PUSH PSW instruction.)

The subroutine is re-entrant and changes all the registers except C.

This subroutine has three parameters — the two starting addresses and the length of the strings. These parameters use five of the seven general-purpose registers.

## Multiple-Precision Addition

**Purpose:** Add two multiple-word binary numbers. The length of the numbers (in bytes) is in the Accumulator, the starting addresses of the numbers are in Registers D and E and H and L, and the starting address of the result is in Registers B and C. All the numbers begin with the least significant bits.

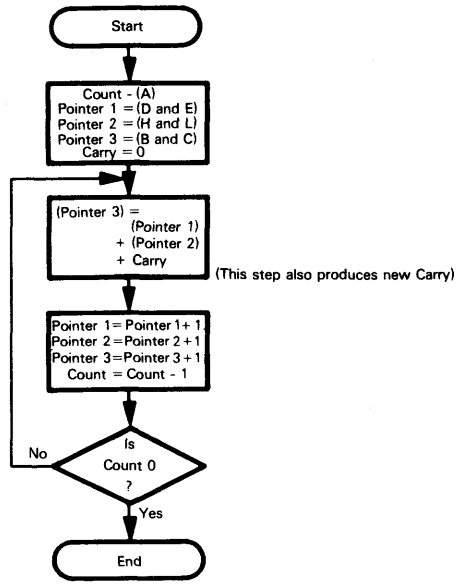
**Sample Problem:**

(A) = 04
(D AND E) = 51
(H AND L) = 61
(B AND C) = 71
(51) = C3
(52) = A7
(53) = 5B
(54) = 2F
(61) = B8
(62) = 35
(63) = DF
(64) = 14
Result = (71) = 7B
(72) = DD
(73) = 3A
(74) = 44

i.e.,

```
  2F5BA7C3
  14DF35B8
----------
  443ADD7B
```

**Flowchart:**



**Source Program:**

The calling program starts the Stack at memory location 80, sets the starting addresses of the various numbers to 50, 60 and 70 respectively, gets the length of the numbers from memory location 40, and calls the multiple-precision addition subroutine.

```
        ORG     0
        LXI     SP,80H      ;START STACK AT LOCATION 80
        LXI     H,50H       ;GET STARTING ADDRESS OF FIRST NUMBER
        LXI     D,60H       ;GET STARTING ADDRESS OF SECOND NUMBER
        LXI     B,70H       ;GET STARTING ADDRESS OF RESULT
        LDA     40H         ;GET LENGTH OF NUMBERS
        CALL    MPADD       ;MULTIPLE-PRECISION ADDITION
HERE:   JMP     HERE
```

The subroutine performs the multiple-precision binary addition.

```
        ORG     20H
MPADD:  PUSH    B           ;RESULT ADDRESS TO STACK
        MOV     B,A
        ANA     A           ;INITIAL CARRY = 0
ADDW:   LDAX    D           ;GET WORD FROM FIRST NUMBER
        ADC     M           ;ADD WORD FROM SECOND NUMBER
        XTHL                ;GET RESULT ADDRESS
        MOV     M,A         ;STORE WORD OF RESULT
        INX     H
        XTHL                ;SAVE RESULT ADDRESS
        INX     D           ;UPDATE POINTERS AND COUNT
        INX     H
        DCR     B
        JNZ     ADDW
        POP     B           ;ELIMINATE RESULT ADDRESS FROM STACK
        RET
```

Subroutine documentation:

```
;
;SUBROUTINE MPADD
;
;PURPOSE: MPADD ADDS TWO
;   MULTIPLE-WORD BINARY NUMBERS
;
;INITIAL CONDITIONS: STARTING ADDRESSES
;   OF NUMBERS IN D AND E, H AND L;
;   STARTING ADDRESS OF RESULT IN
;   B AND C, LENGTH OF NUMBERS IN
;   A
;
;REGISTERS USED: ALL
;
;SAMPLE CASE:
;   STARTING CONDITIONS: (H AND L)
;       = 50, (D AND E) = 60, (B AND C) = 70, (A) = 2,
;       (50) = C3, (51) = A7, (60) = B8, (61) = 35
;   FINAL CONDITIONS: (70) = 7B, (71) = DD
;
```

**Object Program:**

Calling program:

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | SP,80H | 31 |
| 01 | | | | 80 |
| 02 | | | | 00 |
| 03 | | LXI | H,50H | 21 |
| 04 | | | | 50 |
| 05 | | | | 00 |
| 06 | | LXI | D,60H | 11 |
| 07 | | | | 60 |
| 08 | | | | 00 |
| 09 | | LXI | B,70H | 01 |
| 0A | | | | 70 |
| 0B | | | | 00 |
| 0C | | LDA | 40H | 3A |
| 0D | | | | 40 |
| 0E | | | | 00 |
| 0F | | CALL | MPADD | CD |
| 10 | | | | 20 |
| 11 | | | | 00 |
| 12 | HERE: | JMP | HERE | C3 |
| 13 | | | | 12 |
| 14 | | | | 00 |

Subroutine:

| | | | | |
|---|---|---|---|---|
| 20 | MPADD: | PUSH | B | C5 |
| 21 | | MOV | B,A | 47 |
| 22 | | ANA | A | A7 |
| 23 | ADDW: | LDAX | D | 1A |
| 24 | | ADC | M | 8E |
| 25 | | XTHL | | E3 |
| 26 | | MOV | M,A | 77 |
| 27 | | INX | H | 23 |
| 28 | | XTHL | | E3 |
| 29 | | INX | D | 13 |
| 2A | | INX | H | 23 |
| 2B | | DCR | B | 05 |
| 2C | | JNZ | ADDW | C2 |
| 2D | | | | 23 |
| 2E | | | | 00 |
| 2F | | POP | B | C1 |
| 30 | | RET | | C9 |

The instruction XTHL exchanges the top two locations in the Stack with the contents of Registers H and L. This single instruction gets the address for the result from the Stack and saves the data address in the Stack. It allows you to use the top of the Stack as an extra register pair.

The POP B instruction at the end of the program clears the data from the Stack so that the Return instruction will operate properly. You must be careful to clear the Stack correctly since the Return instruction uses the top entry in the Stack as its destination.

This subroutine has four parameters — three addresses and the length of the numbers. All of the registers are used for passing parameters.

The subroutine needs some registers for its own use and must therefore employ some temporary storage. The Stack is not only easy to use for this purpose but also makes the program re-entrant.

## PROBLEMS

Note that you are to write both a calling program for the sample problem and a properly documented subroutine.

### 1). ASCII To Hex

**Purpose:** Convert the contents of the Accumulator from the ASCII representation of a hexadecimal digit to the actual digit itself. Place the result in the Accumulator.

**Sample Problems:**

a.
(A) = 43 'C'
Result = (A) = 0C

b.
(A) = 36 '6'
Result = (A) = 06

### 2) Length Of A Teletype Message

**Purpose:** Determine the length of an ASCII-coded teletype message. The starting address of the string of characters in which the message is embedded is in Registers H and L. The message itself starts with an ASCII STX character (hex 02) and ends with ETX (hex 03). Place the length of the message (the number of characters between the STX and the ETX) in the Accumulator.

**Sample Problem:**

(H AND L) = 41
(41) = 49
(42) = 02 STX
(43) = 47 G
(44) = 4F O
(45) = 03 ETX
Result = (A) = 02

### 3) Check Even Parity In ASCII Characters

**Purpose:** Check the parity of a string of ASCII characters. The length of the string is in the Accumulator and the starting address of the string is in Registers H and L. If the parity of all the characters in the string is even, clear the Accumulator; otherwise, set the Accumulator to FF hex (all 1s).

**Sample Problems:**

a.
(A) = 03
(H AND L) = 42
(42) = B1
(43) = B2
(44) = 33
Result = (A) = 00 since all the characters have even parity.

b.
(A) = 03
(H AND L) = 42
(42) = B1
(43) = B6
(44) = 33
Result = (A) = FF (hex) since the character in memory location 43 does not have even parity.

## 4) String Comparison

**Purpose:** Compare two strings of ASCII characters to see which is larger (i.e., which follows the other in 'alphabetical' ordering). The length of the strings is in the Accumulator; the starting address of string 1 is in Registers H and L and the starting address of string 2 is in Registers D and E. If string 1 is larger than or equal to string 2, clear the Accumulator; otherwise, set the Accumulator to FF hex (all 1s).

**Sample Problems:**

a.

(A) = 03
(D AND E) = 60
(H AND L) = 50
(50) = 43 C
(51) = 41 A
(52) = 54 T
(62) = 42 B
(63) = 41 A
(64) = 54 T
Result = (A) = 00 since CAT is 'larger' than BAT.

b.

(A) = 03
(D AND E) = 60
(H AND L) = 50
(50) = 44 D
(51) = 4F O
(52) = 47 G
(60) = 44 D
(61) = 4F O
(62) = 47 G
Result = (A) = 00 since the two strings are equal.

c.

(A) = 03
(D AND E) = 60
(H AND L) = 50
(50) = 43 C
(51) = 41 A
(52) = 54 T
(60) = 43 C
(61) = 55 U
(62) = 54 T
Result = (A) = FF (hex) since CUT is 'larger' than CAT.

## 5) Decimal Subtraction

**Purpose:** Subtract two multiple-word decimal (BCD) numbers. The length of the numbers (in bytes) is in the Accumulator, the starting addresses of the numbers are in Registers D and E and H and L (subtract the one with the starting address in H and L). The starting address of the result is in Registers B and C. All the numbers begin with the least significant digits. Return the sign of the result in the Accumulator — zero if the result is positive, FF (hex) if it is negative.

**Sample Problem:**

(A) = 04
(H AND L) = 50
(D AND E) = 60
(B AND C) = 70

(50) = 85
(51) = 19
(52) = 70
(53) = 36

(60) = 59
(61) = 34
(62) = 66
(63) = 12

Result = (A) = 00 (positive)
(70) = 26
(71) = 85
(72) = 03
(73) = 24

i.e.,
```
 36701985
-12663459
+24038526
```

# Chapter 11
# INPUT/OUTPUT

There are two problems in the design of input/output systems: one is how to interface peripherals to the computer and transfer data, status and control signals; the other is how to address I/O devices so that the CPU can select a particular one for a data transfer. Clearly, the first problem is both more complex and more interesting. We will therefore discuss the interfacing of peripherals and leave addressing to a more hardware-oriented book.

In theory, the transfer of data to or from an I/O device is not much different than the transfer of data to or from memory. In fact, we can consider the memory as just another I/O device. The memory is, however, rather special for the following reasons:

1) It operates at almost the same speed as the processor.

2) It uses the same type of binary voltage signals as the CPU. The only devices usually needed to interface the memory and the CPU are drivers and receivers.

**I/O AND MEMORY**

3) It requires no special formats or any control signals beyond a READ/WRITE pulse.

4) It automatically latches data sent to it.

5) Its basic word length is the same as that of the computer.

Most I/O devices do not have such convenient features. They may operate at speeds much slower than the processor; e.g., a teletypewriter can transfer only 10 characters per second while a slow processor can transfer 10 000 characters per second. The range of speeds is also very large — sensors may provide one reading per minute while CRTs or floppy disks may transfer 250 000 bits per second. Furthermore, I/O devices may require continuous signals (e.g., motors or thermometers), currents rather than voltages (e.g., a teletypewriter), or far different voltage signals than the signals used by the processor (e.g., gas-discharge displays). I/O devices may also require special formats, protocols or control signals. Their basic word length may be much shorter or much longer than the word length of the computer. All of these variations mean that the design of I/O systems is difficult and has few general principles. Each peripheral is unique and presents its own special interfacing problem.

**I/O CATEGORIES**

We may, however, provide some general description of devices and interfacing methods. We may roughly separate devices into three categories based on their data rates:

1) Slow devices which change state no more than once per second and whose state changes typically last milliseconds or longer. Such devices include lighted displays, switches, relays, and many mechanical sensors and actuators.

2) Medium-speed devices which have data transfer rates of 1 to 10 000 bits per second. Such devices include keyboards, printers, card and paper tape readers and punches, cassette drives, ordinary communications lines and many analog data acquisition systems.

3) High-speed devices which have data transfer rates over 10 000 bits per second. Such devices include magnetic tapes, magnetic disks, high-speed line printers, high-speed communications lines and video displays.

**INTERFACING SLOW DEVICES**

The interfacing of slow devices is relatively simple. Few control signals are necessary except for multiplexing, i.e., handling several devices from one port as shown in Figures 11-1 to 11-4. Input data from slow devices does not require a latch because of the long time constants involved. Output data must, of course, be latched. The only problems that occur while data is being input are transitions at the instant when the input data is being sampled. Hardware circuits or software delay routines can smooth out the transition periods.

A single port can handle several slow devices. Figure 11-1 shows a demultiplexer which automatically directs data to the next consecutive device by counting output operations. Figure 11-2 shows a control port which provides select lines to a demultiplexer. The order here need not be consecutive, but an output instruction is necessary to change the state of the control port. The output demultiplexer is commonly used to drive several displays from the same output port. Figures 11-3 and 11-4 show the same alternatives for an input multiplexer.

Note the differences between input and output with slow devices:

1) Input data need not be latched since the input device provides the data for an enormous length of time by computer standards. Output data must be latched since the output device will not respond to data which is only present for a few CPU clock cycles.

2) Input transitions are a major problem because of their length; output transitions are no problem because of the slow reaction of the output device compared with a CPU clock cycle.

3) The major constraints on input are reaction time and responsiveness; the major constraints on output are response time and observability.

Medium-speed devices must be synchronized in some way to the processor clock. The CPU cannot simply treat these devices as if they held their data forever or could receive data at any time. Instead, the CPU must have a way of discovering when a device is presenting new input data or when a device is ready to receive output data. It must also have a way of telling an output device that new output data is ready.

**HANDSHAKE**

The standard unclocked procedure is the handshake. Here the sender transfers the data and indicates the presence of data to the receiver; the receiver reads the data and completes the handshake by acknowledging the receipt. The receiver may control the situation by initially requesting the data or by indicating its readiness to accept data; the sender then sends the data and completes the handshake by sending an indicator of the presence of data. In either case, the sender knows that the transfer has proceeded successfully and the receiver knows when new data is present.

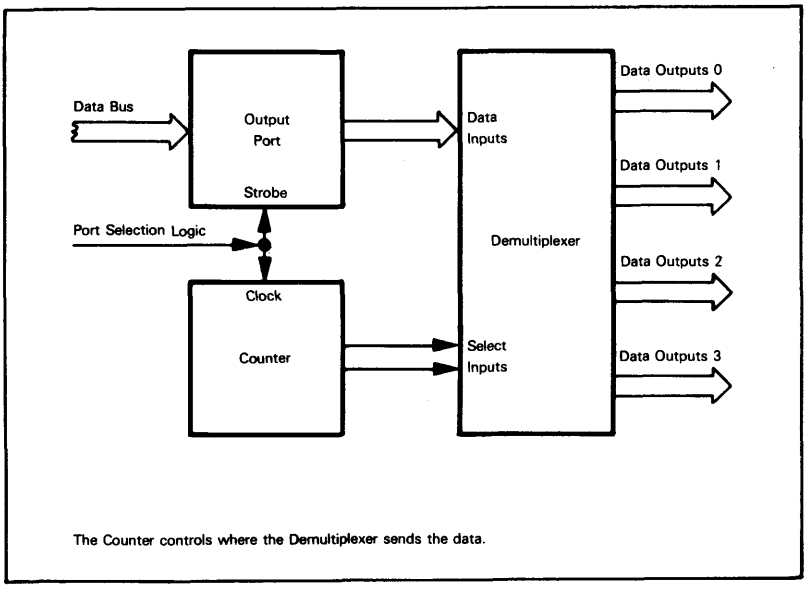



Figure 11-1. An Output Demultiplexer Controlled By A Counter

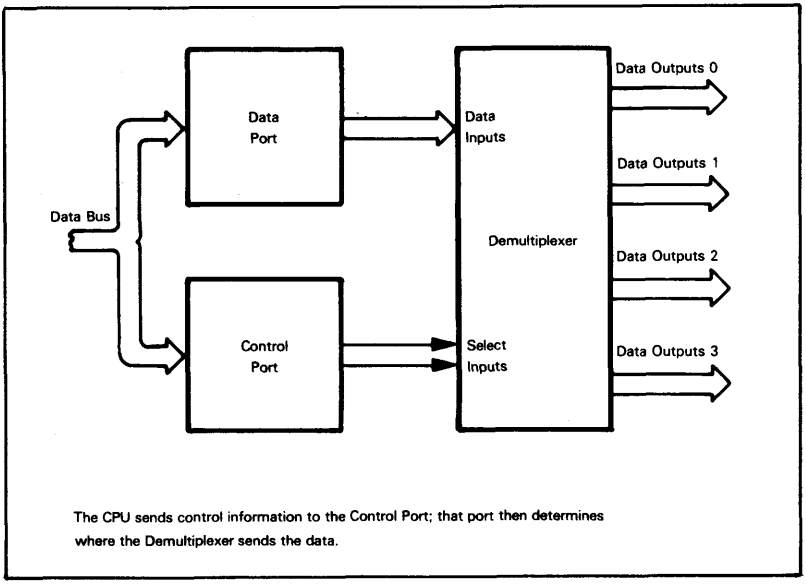



Figure 11-2. An Output Demultiplexer Controlled By A Port

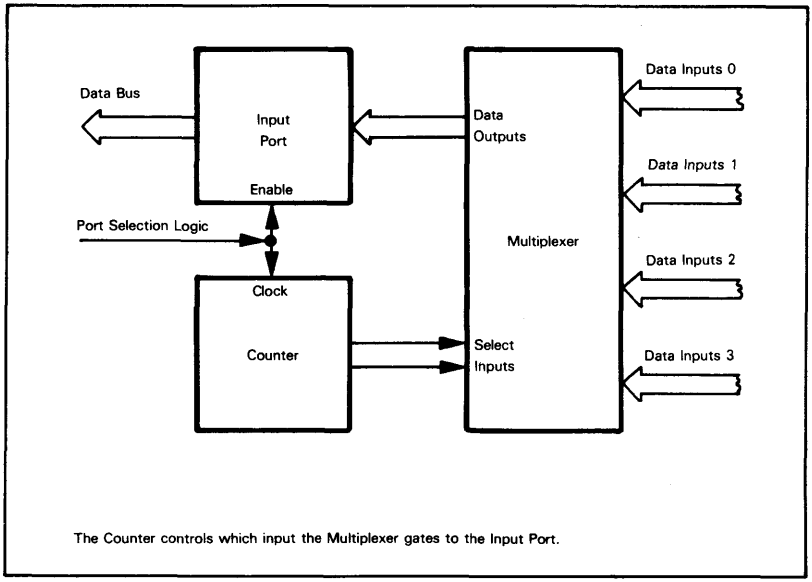



Figure 11-3. An Input Multiplexer Controlled By A Counter

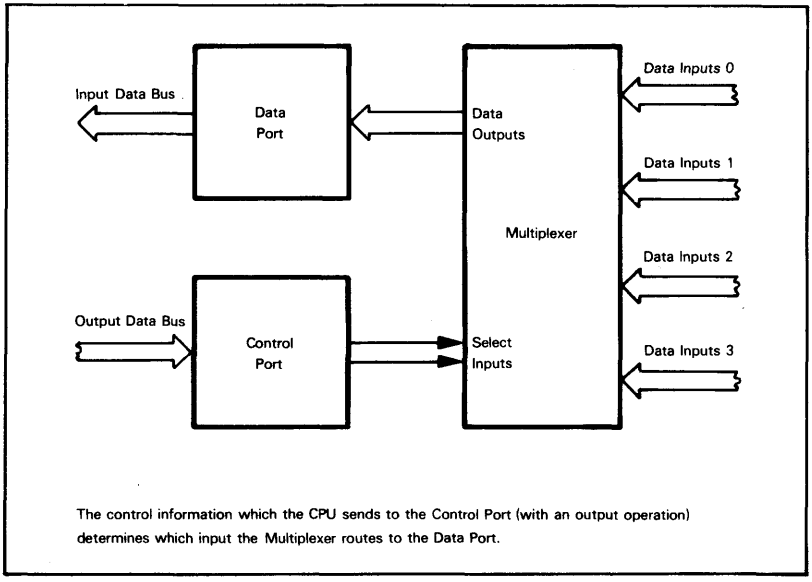



Figure 11-4. An Input Multiplexer Controlled By A Port

Figures 11-5 and 11-6 show typical input and output operations using the handshake method. The procedure whereby the CPU performs an input operation to check the readiness of the peripheral before transferring data is called "polling". Clearly, polling can occupy a large amount of processor time if there are many I/O devices. There are several ways of providing the handshake signals. Among these are:

- Separate dedicated I/O lines. The processor may handle these through its regular I/O system or through special lines or interrupts. The 8085 processor has a special serial input line (SID) and a special serial output line (SOD); the 8080 processor does not have these lines.
- Special patterns on the I/O lines. These may be single start and stop bits or entire characters or groups of characters. The patterns must be easily distinguished from background noise or the ordinary state of the lines.

**STROBE**

We often call a separate I/O line which indicates the presence of data or the occurrence of an operation a "strobe". A strobe may, for example, place data in a latch or fetch it from a buffer.

Many peripherals transfer data at regular intervals, i.e., synchronously. Here the only problem is starting the process, i.e., lining up to the first input or marking the first output. In some cases the first transfer proceeds asynchronously; in other cases the peripheral provides a clock signal which the processor can examine for timing purposes.

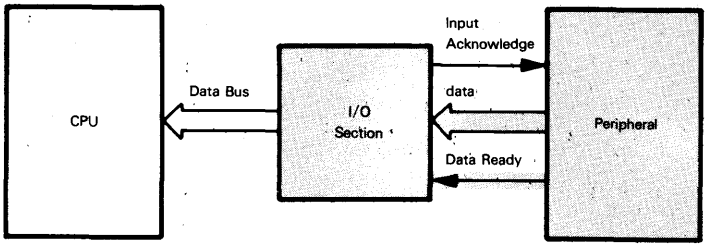
**REDUCING TRANSMISSION ERRORS**

One problem with medium-speed devices is errors in transmission. Several methods can lessen the likelihood of such errors; they include:

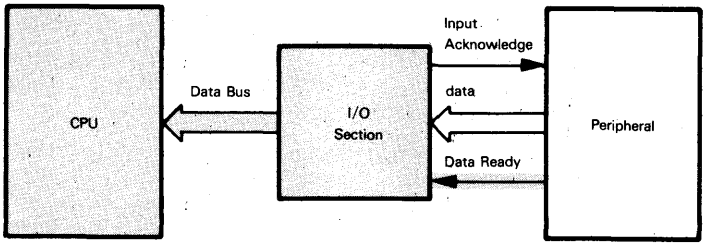
- Sampling input data at the center of the transmission interval to avoid transition effects
- Sampling each input several times and using majority logic, e.g., if there are five input lines and three or more of them are 'on', then the signal is assumed to be 'on'
- Generating and checking parity, i.e., an extra bit which makes the number of 1 bits in the correct data even or odd
- Using other error detecting and correcting codes such as checksums, LRC (longitudinal redundancy check), and CRC (cyclic redundancy check)

**DIRECT MEMORY ACCESS**

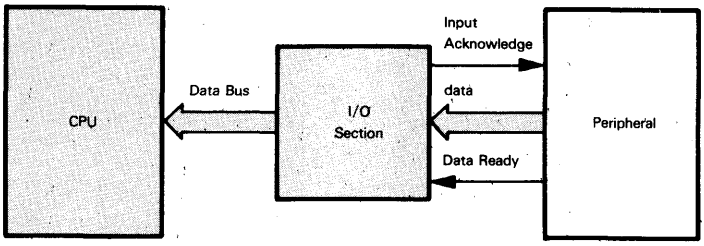
High-speed devices which transfer more than 10 000 bits per second require special methods. The usual technique is to construct a special-purpose external controller which transfers data directly between the memory and an I/O device. This process is called direct memory access (DMA). The DMA controller must force the CPU off the busses, provide addresses and control signals to the memory, and transfer the data. Such a controller will be fairly complex, typically consisting of 50 to 100 chips, although LSI devices are now available. For example, the 8257 Direct Memory Access Controller for 8080-based microcomputers is described in An Introduction To Microcomputers: Volume II — Some Real Products. The CPU must initially load the Address and Data Counters in the controller so that the controller will know where to start and how much data to transfer.



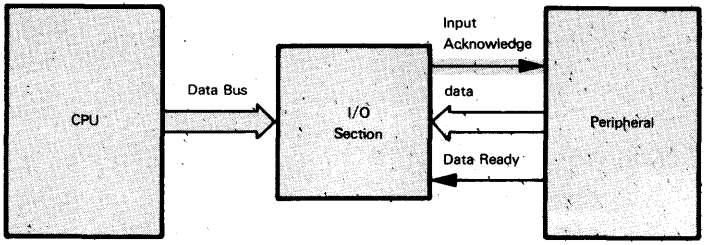

a) Peripheral provides data and Data Ready signal to computer I/O section.




b) CPU reads Data Ready signal from I/O section (this may be a hardware, e.g., interrupt connection).

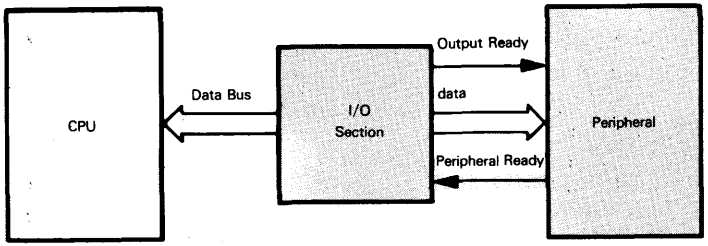



c) CPU reads data from I/O section.

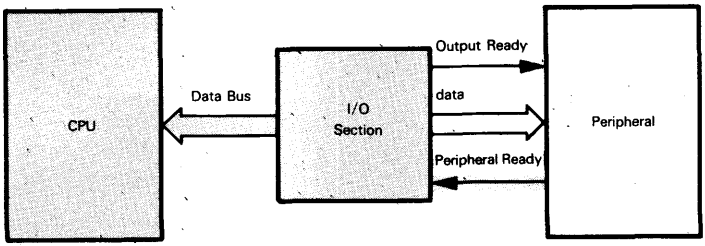



d) CPU sends Input Acknowledge signal to I/O section which then provides Input Acknowledge signal to Peripheral (this may be a hardware connection).

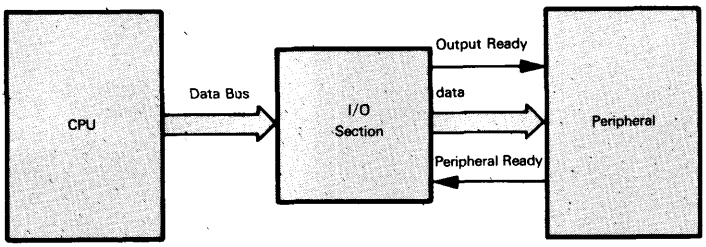
Figure 11-5. An Input Handshake



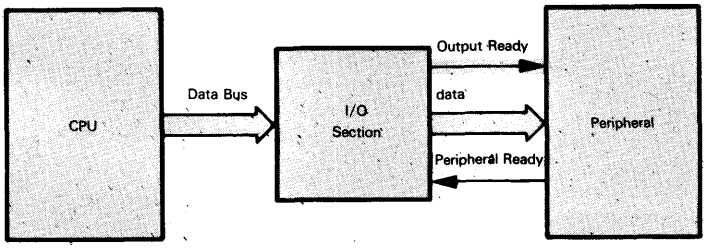

a) Peripheral provides Peripheral Ready signal to computer I/O section.




b) CPU reads Peripheral Ready signal from I/O section (this may be a hardware, e.g., interrupt connection).




c) CPU sends data to Peripheral.




d) CPU sends Output Ready signal to Peripheral (this may be a hardware connection).

Figure 11-6. An Output Handshake

## TIMING INTERVALS (DELAYS)

**USES OF TIMING INTERVALS**

One problem which we will face throughout the discussion of input/output is the generation of timing intervals of a specific length. Such intervals are necessary to debounce mechanical switches (i.e., to smooth their irregular transitions), to provide pulses with specified lengths and frequencies for displays, and to provide timing for devices that either send or receive data regularly (e.g., a teletypewriter which sends or receives one bit every 9.1 ms).

**METHODS FOR PRODUCING TIMING INTERVALS**

We can produce timing intervals in several ways:

1) In hardware with one-shots (monostable multivibrators). These devices produce a single pulse of fixed duration in response to a trigger input.

2) In a combination of hardware and software with a flexible programmable timer, such as the 8253 Programmable Timer for 8080-based microcomputers as described in An Introduction To Microcomputers: Volume II — Some Real Products. The 8253 can provide timing intervals of various lengths with a variety of starting and ending conditions.

3) In software with delay routines. These routines use the processor as a counter. This use is possible if the processor has a stable clock reference, but it clearly under-utilizes the processor. However, delay routines require no additional hardware and often use processor time which would otherwise be wasted.

**CHOOSING A TIMING METHOD**

The choice among these three methods depends on your application. The software method is inexpensive but may overburden the processor. The programmable timers are relatively expensive, but are easy to interface and may be able to handle many complex timing tasks. The use of one-shots is usually avoided by good digital designers.

## DELAY ROUTINES

**BASIC SOFTWARE DELAY**

A simple delay routine works as follows:

STEP 1 - Load a register with a specified value.

STEP 2 - Decrement the register.

STEP 3 - If the result of STEP 2 is not zero, repeat STEP 2.

This routine is used strictly for timing purposes, it serves no other function. The amount of time used depends upon the execution time of the various instructions and the specified value loaded into the register. The maximum length of the delay is limited by the size of the register; however, the entire routine can be placed inside a similar routine which uses another register and so on.

**TRANSPARENT DELAY ROUTINE**

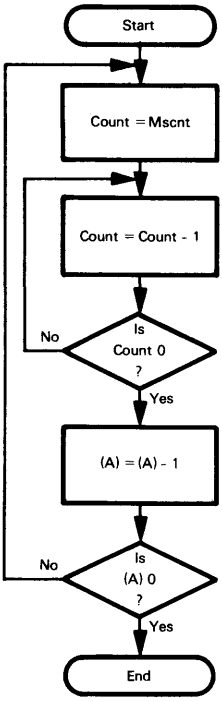
The following example uses Register C and the Accumulator to provide delays as long as 255 ms. The choice of registers is arbitrary. You may, in fact, find the use of a register pair (e.g., B and C) more convenient. A PUSH B instruction at the start of the delay routine and a POP B at the end will result in a routine that does not affect any registers at all. Note that the PUSH and POP instructions must be included in the time budget.

## EXAMPLES

### Delay Program

**Purpose:** The program provides a delay of 1 ms times the contents of the Accumulator.

**Flowchart:**




The value of MSCNT depends on the speed of the CPU and the memory cycle.

**Source Program:**

```
DELAY:  MVI     B,MSCNT ;GET COUNT FOR 1 MS DELAY
DLY1:   DCR     B       ;COUNT=COUNT - 1
        JNZ     DLY1    ;CONTINUE UNTIL COUNT=0
        DCR     A       ;NUMBER OF MS=NUMBER OF MS - 1
        JNZ     DELAY   ;CONTINUE UNTIL NUMBER OF MS=0
        RET
```

**Object Program:** (starting in location 30)

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 30 | DELAY: | MVI | B,MSCNT | 06 |
| 31 | | | | MSCNT |
| 32 | | DCR | B | 05 |
| 33 | | JNZ | DLY1 | C2 |
| 34 | | | | 32 |
| 35 | | | | 00 |
| 36 | | DCR | A | 3D |
| 37 | | JNZ | DELAY | C2 |
| 38 | | | | 30 |
| 39 | | | | 00 |
| 3A | | RET | | C9 |

**Time Budget:**

| Instruction | | Number Of Times Executed |
|---|---|---|
| MVI | B,MSCNT | (A) |
| DCR | B | (A) x MSCNT |
| JNZ | DLY1 | (A) x MSCNT |
| DCR | A | (A) |
| JNZ | DELAY | (A) |
| RET | | |

The total time used should be (A) x 1 ms. If the memory is operating at full speed, the instructions take the following number of clock cycles:

| | | |
|---|---|---|
| MVI | B | 7 |
| DCR | | 5 |
| JNZ | | 10 |

Ignoring the CALL and RET instructions (which only occur once), the program takes

(A) x (7+15 x MSCNT+15)

clock cycles. So, to make the delay 1 ms,

$22+(15 \times MSCNT) = N_C$

where $N_C$ is the number of clock cycles per millisecond. At the standard 2 MHz 8080 clock rate, $N_C = 2000$, so

$15 \times MSCNT = 1978$

MSCNT=132 (hex 84) at an 8080 clock rate of 2 MHz

**8080A DELAY LOOP CONSTANT**

The 8085 timing is different since DCR takes 4 clock cycles on that processor. Furthermore, JNZ requires only 7 clock cycles when no Jump is performed.

**8085 DELAY LOOP CONSTANT**

So, to produce a 1 ms delay on the 8085, the equation is:

$$7+14 \times MSCNT - 3+11 = N_C$$

or

$$(14 \times MSCNT)+15 = N_C$$

The -3 is caused by the fact that the last unsuccessful JNZ instruction in the inner loop uses 7 rather than 10 cycles. At the standard 3 MHz 8085 clock rate, $N_C = 3000$, so

$$14 \times MSCNT = 2985$$

MSCNT = 213 (hex D5) at an 8085 clock rate of 3 MHz

## A Pushbutton (Or SPST Momentary Switch)

**Purpose:** To interface one pushbutton switch (or a single-pole, single-throw momentary switch) to an 8080 microprocessor. The pushbutton is a mechanical switch which provides a single contact closure (i.e., a logic '0') while pushed.

**Circuit Diagram:**

Figure 11-7 shows the circuitry required to interface the pushbutton. It uses one bit of an 8212 input port which acts as a buffer: no latch is needed since the pushbutton closure is present for a very long time by CPU standards. (We will not discuss the use of the 8212 port in detail in this book. You can find a complete description of the device in Volume II of An Introduction To Microcomputers).

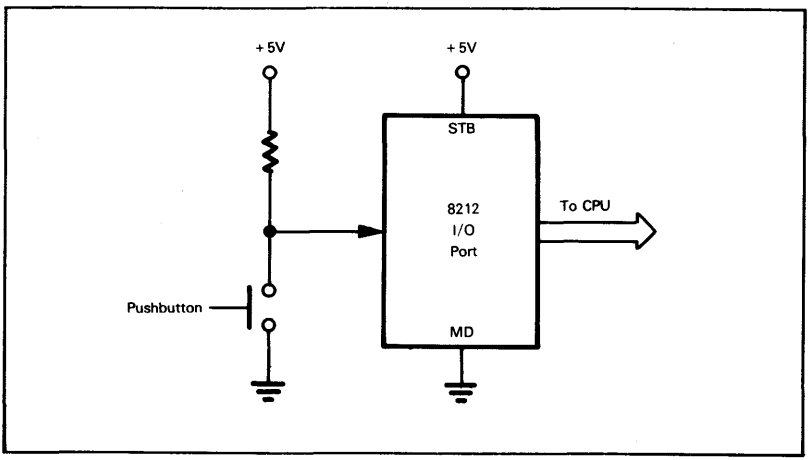



Figure 11-7. A Pushbutton Circuit

**Programming Examples:**

We will perform two tasks that involve this circuit. They are:

1) Set a memory location based on the value of the switch.
2) Count the number of times that the switch is closed.

**Task 1:** Set memory location 40 to 1 if the button is open ('1'), 0 if it is closed ('0').

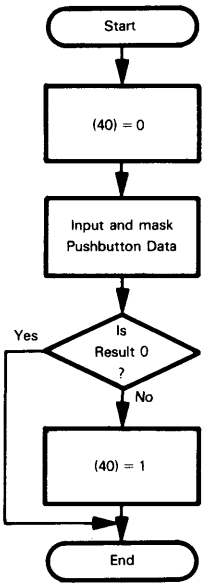
**Sample Problems:**

a.

Button open (i.e., not pushed)
Result = (40) = 01

b. Button closed (i.e., pushed)
Result = (40) = 00

**Flowchart:**




**Source Program:**

```
        LXI     H,40H
        MVI     M,0     ;MARKER=0
        IN      PORT    ;READ BUTTON POSITION
        ANI     MASK    ;IS BUTTON CLOSED (0)?
        JZ      DONE    ;YES, DONE
        INR     M       ;NO, MARKER=1
DONE:   JMP     DONE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MVI | M,0 | 36 |
| 04 | | | | 00 |
| 05 | | IN | PORT | DB |
| 06 | | | | PORT |
| 07 | | ANI | MASK | E6 |
| 08 | | | | MASK |
| 09 | | JZ | DONE | CA |
| 0A | | | | 0D |
| 0B | | | | 00 |
| 0C | | INR | M | 34 |
| 0D | DONE: | JMP | DONE | C3 |
| 0E | | | | 0D |
| 0F | | | | 00 |

The port number depends on which port the pushbutton is connected to. MASK depends on which bit of the port the pushbutton is connected to; MASK has a '1' bit in the button position and zeros elsewhere.

| Button Position (Bit Number) | Mask | |
|---|---|---|
| | Binary | Hex |
| 0 | 00000001 | 01 |
| 1 | 00000010 | 02 |
| 2 | 00000100 | 04 |
| 3 | 00001000 | 08 |
| 4 | 00010000 | 10 |
| 5 | 00100000 | 20 |
| 6 | 01000000 | 40 |
| 7 | 10000000 | 80 |

If the button is attached to bit 0 or bit 7 of the input port, the program can use a Shift instruction to set the Carry and thereby determine the button's state. For example:

Bit 7

```
IN      PORT    ;READ BUTTON POSITION
RAL             ;IS BUTTON CLOSED (0)?
JNC     DONE    ;YES, DONE
```

Bit 0

```
IN      PORT    ;READ BUTTON POSITION
RAR             ;IS BUTTON CLOSED (0)?
JNC     DONE    ;YES, DONE
```

If the button is attached to bits 6 or 7 of the input port, the program can use the Sign bit to determine the button's state. For example:

Bit 7

```
IN      PORT    ;READ BUTTON POSITION
ORA     A       ;IS BUTTON CLOSED (0)?
JP      DONE    ;YES, DONE
```

IN does not affect the flags; therefore we must use the ORA instruction to set the flags.

Bit 6

```
IN      PORT    ;READ BUTTON POSITION
ADD     A       ;IS BUTTON CLOSED (0)?
JP      DONE    ;YES, DONE
```

RAL cannot be used because it does not affect the Sign bit.

**Task 2:** Count the number of button closures by incrementing memory location 40.

**SWITCH BOUNCE**

In order to count the number of times the button has been pressed, we must be sure that each closure causes a single transition. However, a mechanical pushbutton does not produce a single transition for each closure because the mechanical contacts bounce back and forth before settling in their final positions. We can use hardware to eliminate the bounce or we can handle it in software.

**DEBOUNCING IN SOFTWARE**

The program can debounce the pushbutton by waiting after it finds a closure. The required delay is called the debouncing time and is a characteristic of the pushbutton. It is typically a few milliseconds. The program should not examine the pushbutton during this period because it might mistake the bounce for a new closure. The program may either enter a delay routine like the one described previously or may simply perform other tasks for the specified amount of time.

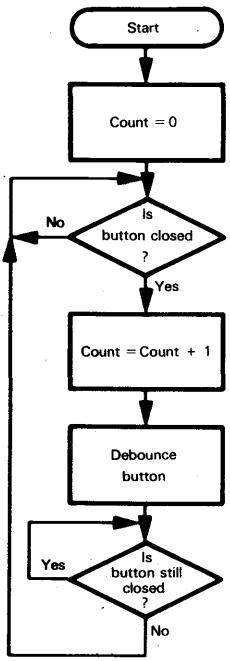
Even after debouncing, the program must still wait for the present closure to end before looking for a new closure. This procedure avoids counting the same closure more than once. The following program uses a software delay of 1 ms to debounce the pushbutton. You may want to try varying the delay or eliminating it entirely in order to see what happens. To run this program you must also enter the delay subroutine into memory, starting at location 30.

**Sample Problem:**

Pressing the button ten times after the start of the program should give

(40) = 0A

**Flowchart:**




**Source Program:**

```
        LXI     H,40H     ;COUNT=0
        MVI     M,0
CHKCL:  IN      PORT      ;IS BUTTON CLOSED (0)?
        ANI     MASK
        JNZ     CHKCL     ;NO, WAIT
        INR     M         ;YES, COUNT=COUNT+1
        MVI     A,1
        CALL    DELAY     ;DEBOUNCE BUTTON BY WAITING FOR 1 MS
CHKOP:  IN      PORT      ;IS BUTTON STILL CLOSED (0)?
        ANI     MASK
        JZ      CHKOP     ;YES, WAIT
        JMP     CHKCL     ;NO, LOOK FOR NEXT CLOSURE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MVI | M,0 | 36 |
| 04 | | | | 00 |
| 05 | CHKCL: | IN | PORT | DB |
| 06 | | | | PORT |
| 07 | | ANI | MASK | E6 |
| 08 | | | | MASK |
| 09 | | JNZ | CHKCL | C2 |
| 0A | | | | 05 |
| 0B | | | | 00 |
| 0C | | INR | M | 34 |
| 0D | | MVI | A,1 | 3E |
| 0E | | | | 01 |
| 0F | | CALL | DELAY | CD |
| 10 | | | | 30 |
| 11 | | | | 00 |
| 12 | CHKOP: | IN | PORT | DB |
| 13 | | | | PORT |
| 14 | | ANI | MASK | E6 |
| 15 | | | | MASK |
| 16 | | JZ | CHKOP | CA |
| 17 | | | | 12 |
| 18 | | | | 00 |
| 19 | | JMP | CHKCL | C3 |
| 1A | | | | 05 |
| 1B | | | | 00 |

Note that the three instructions beginning with the label CHKOP are used to determine when the switch reopens.

## A Toggle (SPDT) Switch

**Purpose:** To interface a single-pole, double-throw switch to an 8080 microprocessor. The toggle switch is a mechanical device which is either in the normally closed (NC) position or the normally open (NO) position.

**DEBOUNCING WITH CROSS-COUPLED NAND GATES**

Figure 11-8 shows the circuitry required to interface the switch. Like the pushbutton, the switch uses one bit of an unlatched 8212 input port which serves as an addressable buffer. Unlike the button, the switch may be left in either position; typical program tasks are to determine the switch position and to see if the position has changed. A pair of cross-coupled NAND gates (see Figure 11-9) can debounce a mechanical switch.

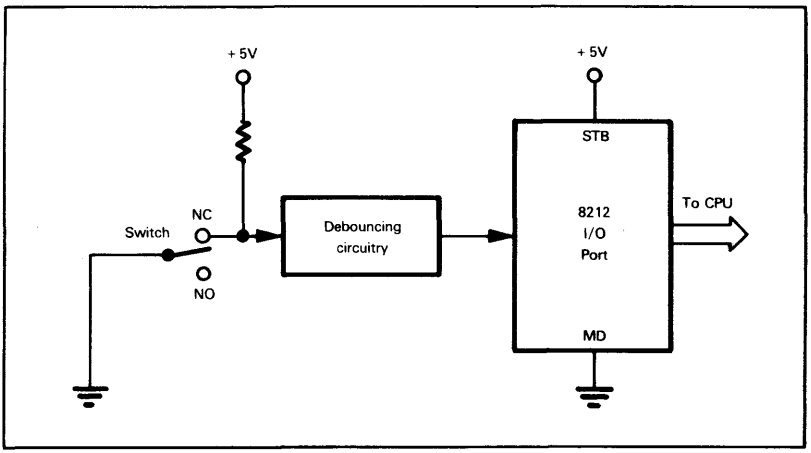



Figure 11-8. A Toggle Switch Circuit

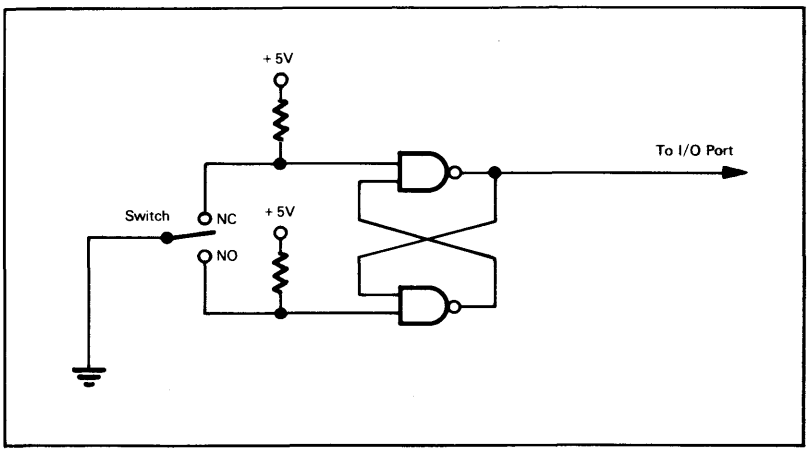



Figure 11-9. A Debouncing Circuit Based On Cross-Coupled NAND Gates

This circuit produces a single step in response to a change in switch position even if the switch bounces before settling into its new position.

**Programming Examples:**

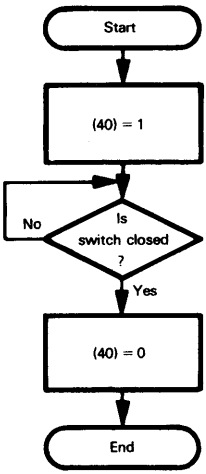
We will perform two tasks that involve this circuit. They are:

1) Monitor the output of the switch and clear a memory location when the normally open contacts of the switch are closed.
2) Monitor the output of the switch and clear a memory location when the switch output changes.

**Task 1:** Wait for switch to close.

Memory location 40 remains 1 until the switch is closed and then is cleared, i.e., the processor sets memory location 40 to 1, waits for the switch to be closed, and then clears memory location 40. The switch acts as a RUN/HALT command since the processor will not proceed until the switch is closed.

**Flowchart:**




**Source Program:**

```
        LXI     H,40H     ;MARKER=1
        MVI     M,1
WAITC:  IN      PORT      ;READ SWITCH POSITION
        ANI     MASK      ;IS SWITCH CLOSED (0)?
        JNZ     WAITC     ;NO, WAIT
        DCR     M         ;YES, MARKER=0
DONE:   JMP     DONE
```

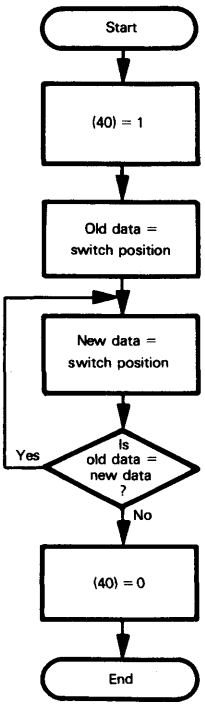
**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MVI | M,1 | 36 |
| 04 | | | | 01 |
| 05 | WAITC: | IN | PORT | DB |
| 06 | | | | PORT |
| 07 | | ANI | MASK | E6 |
| 08 | | | | MASK |
| 09 | | JNZ | WAITC | C2 |
| 0A | | | | 05 |
| 0B | | | | 00 |
| 0C | | DCR | M | 35 |
| 0D | DONE: | JMP | DONE | C3 |
| 0E | | | | 0D |
| 0F | | | | 00 |

**Task 2:** Wait for switch to change.

Memory location 40 remains 1 until the switch position changes, i.e., the processor waits until the switch changes, then clears memory location 40.

**Flowchart:**



**Source Program:**

```
        LXI     H,40H
        MVI     M,1     ;MARKER=1
        IN      PORT    ;GET OLD SWITCH POSITION
        ANI     MASK
        MOV     B,A
SRCH:   IN      PORT    ;GET NEW SWITCH POSITION
        ANI     MASK
        CMP     B       ;ARE NEW AND OLD POSITIONS THE SAME?
        JZ      SRCH    ;YES, WAIT
        DCR     M       ;NO, MARKER=0
DONE:   JMP     DONE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MVI | M,1 | 36 |
| 04 | | | | 01 |
| 05 | | IN | PORT | DB |
| 06 | | | | PORT |
| 07 | | ANI | MASK | E6 |
| 08 | | | | MASK |
| 09 | | MOV | B,A | 47 |
| 0A | SRCH: | IN | PORT | DB |
| 0B | | | | PORT |
| 0C | | ANI | MASK | E6 |
| 0D | | | | MASK |
| 0E | | CMP | B | B8 |
| 0F | | JZ | SRCH | CA |
| 10 | | | | 0A |
| 11 | | | | 00 |
| 12 | | DCR | M | 35 |
| 13 | DONE: | JMP | DONE | C3 |
| 14 | | | | 13 |
| 15 | | | | 00 |

SUB B or XRA B could replace CMP B in the program. Both of these instructions would, however, change the contents of the Accumulator. If several switches were attached to the PORT, then XRA B might be more useful than CMP B, since XRA B would produce a 1 bit for each switch that changed state. How would you rewrite this program in order to debounce the switch in software?

## A Multiple-Position (Rotary, Selector, Or Thumbwheel) Switch

**Purpose:** To interface a multiple-position switch to an 8080 microprocessor. The switch provides a connection to ground in the lead corresponding to the switch position.

Figure 11-10 shows the circuitry required to interface an 8-position switch. The switch uses all eight bits of an 8212 input port. Typical tasks are to determine the position of the switch and to check if that position has changed. Two special situations must be handled:

1) The switch is temporarily between two positions so that no leads are grounded.

2) The switch has not yet reached its final position.

The first of these situations can be handled by waiting until the input is not all 1s, i.e., until some switch position is grounded. We can handle the second situation by examining the switch again after a delay (e.g., 1 or 2 seconds) and only accepting the input when it is repeated. This delay will not usually affect the responsiveness of the system to the switch. We can also use another switch (i.e., a LOAD switch) to tell the processor that the selector switch should be read.

**Programming Examples:**

We will perform two tasks which involve the circuit of Figure 11-10. These are:

1) Monitor the switch output until a stable condition is detected, then determine the position of the switch and store the binary equivalent of the switch position in a memory location.

2) Wait for the position of the switch to change, then store the new switch position in a memory location.

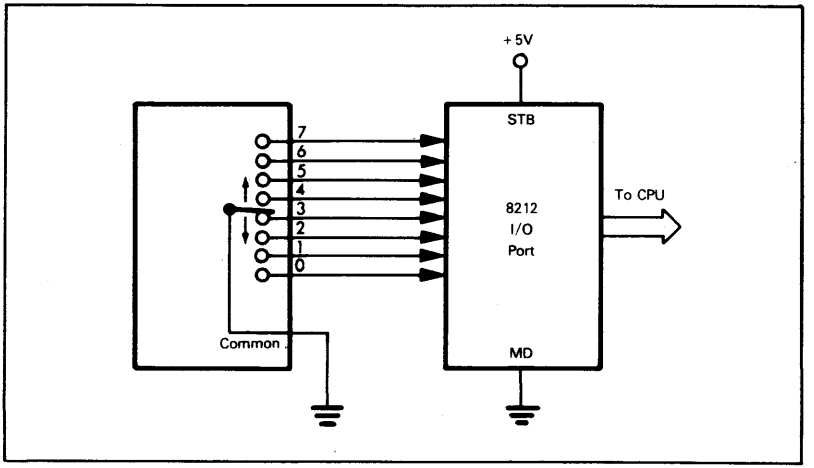



Figure 11-10. A Multiple-Position Switch

If the switch is in a particular position, the lead from that position is grounded through the common line. Pullup resistors are usually necessary on all the input lines to avoid problems caused by noise.

**Task 1:** Determine switch position.

The program waits for the switch to be in a specific position and then places the number of that position in memory location 40.

The following table relates the data input to the switch position for Figure 11-10.

Table 11-1. Data Input vs. Switch Position

| Switch Position | Data Input | |
|---|---|---|
| | Binary | Hex |
| 0 | 11111110 | FE |
| 1 | 11111101 | FD |
| 2 | 11111011 | FB |
| 3 | 11110111 | F7 |
| 4 | 11101111 | EF |
| 5 | 11011111 | DF |
| 6 | 10111111 | BF |
| 7 | 01111111 | 7F |

Note the inefficiency of this scheme, which requires eight bits to distinguish among eight different positions.

**USING A DIGITAL ENCODER**

A digital encoder could reduce the number of bits needed. Figure 11-11 shows a circuit using the 74LS148 TTL 8-to-3 encoder. We attach the switch outputs in inverse order since the 74LS148 device has active-low outputs. The result of the encoder circuit is a 3-bit representation of the switch position. Many switches have encoders or diode matrices included so that they automatically produce a coded output, usually BCD.

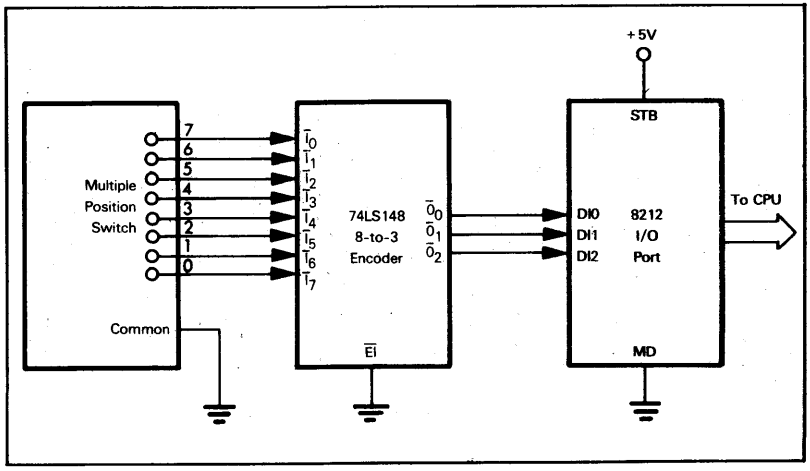
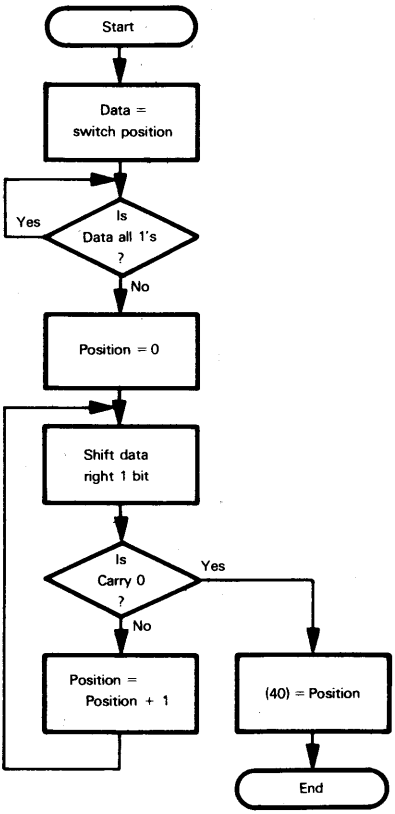



Figure 11-11. A Multiple-Position Switch With An Encoder

The encoder provides outputs that are active-low. Note, for example, that the output from position 5 on the multiple position switch is connected to $\overline{I}_2$ on the encoder. The encoder takes the three-bit binary representation (000) of 2 and produces 101, which is the active-low representation of 2.

**Flowchart:**




**Source Program:**

```
CHKSW:  IN      PORT        ;GET SWITCH DATA
        CPI     11111111B   ;IS SWITCH IN A POSITION?
        JZ      CHKSW       ;NO, WAIT FOR POSITION
        MVI     B,0         ;POSITION = 0
CHPOS:  RAR                 ;IS NEXT BIT GROUNDED POSITION?
        JNC     DONE        ;YES, SWITCH POSITION FOUND
        INR     B           ;NO, POSITION = POSITION+1
        JMP     CHPOS
DONE:   LXI     H,40H       ;STORE SWITCH POSITION
        MOV     M,B
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | CHKSW: | IN | PORT | DB |
| 01 | | | | PORT |
| 02 | | CPI | 11111111B | FE |
| 03 | | | | FF |
| 04 | | JZ | CHKSW | CA |
| 05 | | | | 00 |
| 06 | | | | 00 |
| 07 | | MVI | B,0 | 06 |
| 08 | | | | 00 |
| 09 | CHPOS: | RAR | | 1F |
| 0A | | JNC | DONE | D2 |
| 0B | | | | 11 |
| 0C | | | | 00 |
| 0D | | INR | B | 04 |
| 0E | | JMP | CHPOS | C3 |
| 0F | | | | 09 |
| 10 | | | | 00 |
| 11 | DONE: | LXI | H,40H | 21 |
| 12 | | | | 40 |
| 13 | | | | 00 |
| 14 | | MOV | M,B | 70 |
| 15 | HERE: | JMP | HERE | C3 |
| 16 | | | | 15 |
| 17 | | | | 00 |

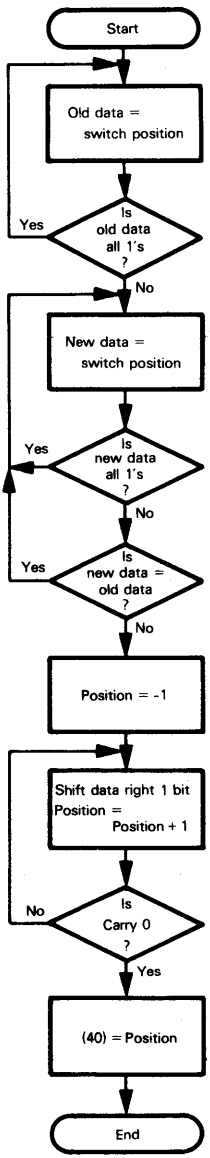
Suppose that a faulty switch or defective I/O port results in the input always being $0FF_{16}$. How could you change the program so that it would detect this error?

There is an unconditional Jump, JMP CHPOS, in the source program. Can you restructure the initial conditions so that this unnecessary jump is eliminated?

**Task 2:** Wait for switch position to change.

The program waits for the switch position to change and places the new position (decoded) in memory location 40. The program ignores intermediate states where the switch is not in any position.

**Flowchart:**



**Source Program:**

```
CHFST:  IN      PORT        ;GET SWITCH DATA
        CPI     11111111B   ;IS SWITCH IN A POSITION?
        JZ      CHFST       ;NO, WAIT FOR POSITION
        MOV     B,A
CHSEC:  IN      PORT        ;GET NEW SWITCH DATA
        CPI     11111111B   ;IS SWITCH IN A POSITION?
        JZ      CHSEC       ;NO, WAIT FOR POSITION
        CMP     B           ;IS POSITION SAME AS BEFORE?
        JZ      CHSEC       ;YES, WAIT
        MVI     B,0FFH      ;POSITION = -1
CHPOS:  RAR                 ;IS NEXT BIT GROUNDED POSITION?
        INR     B           ;POSITION = POSITION+1
        JC      CHPOS       ;NO, KEEP LOOKING FOR GROUNDED POSITION
DONE:   LXI     H,40H       ;STORE SWITCH POSITION
        MOV     M,B
HERE:   JMP     HERE
```

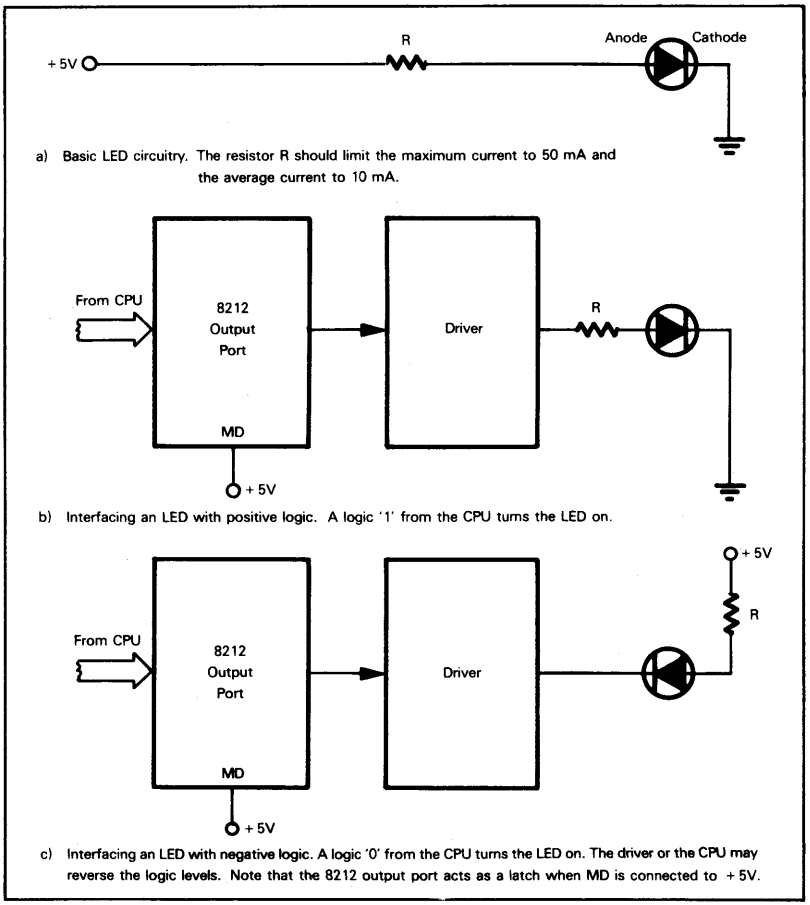
**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | CHFST: | IN | PORT | DB |
| 01 | | | | PORT |
| 02 | | CPI | 11111111B | FE |
| 03 | | | | FF |
| 04 | | JZ | CHFST | CA |
| 05 | | | | 00 |
| 06 | | | | 00 |
| 07 | | MOV | B,A | 47 |
| 08 | CHSEC: | IN | PORT | DB |
| 09 | | | | PORT |
| 0A | | CPI | 11111111B | FE |
| 0B | | | | FF |
| 0C | | JZ | CHSEC | CA |
| 0D | | | | 08 |
| 0E | | | | 00 |
| 0F | | MVI | B,0FFH | 06 |
| 10 | | | | FF |
| 11 | CHPOS: | RAR | | 1F |
| 12 | | INR | B | 04 |
| 13 | | JC | CHPOS | DA |
| 14 | | | | 11 |
| 15 | | | | 00 |
| 16 | DONE: | LXI | H,40H | 21 |
| 17 | | | | 40 |
| 18 | | | | 00 |
| 19 | | MOV | M,B | 70 |
| 1A | HERE: | JMP | HERE | C3 |
| 1B | | | | 1A |
| 1C | | | | 00 |

## A Single LED

**Purpose:** To interface a single light-emitting diode to an 8080 microprocessor. The LED can be attached so that either a '0' or a '1' turns it on.

Figure 11-12 shows the circuitry required to illuminate an LED. The LED lights when its anode is positive with respect to its cathode (Figure 11-12a). So, the circuitry can either light the LED by grounding the cathode and having the computer supply a '1' to the anode (Figure 11-12b), or by tying the anode to +5 volts and having the computer supply a '0' to the cathode (Figure 11-12c). Using the cathode is the most common approach. The LED is brightest when it operates from pulsed currents of about 10 to 50 mA applied a few hundred times per second. LEDs have a very short turn-on time (in the microsecond range), so they are well-suited to multiplexing, i.e., operating several from a single port. LED circuits usually need peripheral or transistor drivers and current-limiting resistors.




a) Basic LED circuitry. The resistor R should limit the maximum current to 50 mA and the average current to 10 mA.

b) Interfacing an LED with positive logic. A logic '1' from the CPU turns the LED on.

c) Interfacing an LED with negative logic. A logic '0' from the CPU turns the LED on. The driver or the CPU may reverse the logic levels. Note that the 8212 output port acts as a latch when MD is connected to +5V.

Figure 11-12. Interfacing an LED

**Programming Examples:**

The program turns a single LED either on or off.

**Task 1:** Send a logic '1' to the LED (i.e., turn a positive display on or a negative display off).

**Source Program:** (form data initially)

```
        MVI     A,MASKP
        OUT     PORT        ;DATA TO LEDS
        STA     40H         ;SAVE COPY OF OUTPUT DATA
HERE:   JMP     HERE
```

(update data in location 40H)

```
        LXI     H,40H       ;LOAD POINTER TO DATA
        MOV     A,M         ;GET COPY OF OUTPUT DATA
        ORI     MASKP       ;LED BIT = 1
        OUT     PORT        ;DATA TO LEDS
        MOV     M,A         ;SAVE COPY OF OUTPUT DATA
HERE:   JMP     HERE
```

MASKP has a '1' in the bit position assigned to the LED we wish to illuminate. There are zeros in all other bit positions. Logically ORing with MASKP does not affect the other bit positions. Note that the CPU cannot directly read the data from the output port, so a copy is necessary.

**Object Program:** (form data initially)

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | MVI | A,MASKP | 3E |
| 01 | | | | MASKP |
| 02 | | OUT | PORT | D3 |
| 03 | | | | PORT |
| 04 | | STA | 40H | 32 |
| 05 | | | | 40 |
| 06 | | | | 00 |
| 07 | HERE: | JMP | HERE | C3 |
| 08 | | | | 07 |
| 09 | | | | 00 |

**Object Program:** (update data in location 40H)

| | | | | |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 21 |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MOV | A,M | 7E |
| 04 | | ORI | MASKP | F6 |
| 05 | | | | MASKP |
| 06 | | OUT | PORT | D3 |
| 07 | | | | PORT |
| 08 | | MOV | M,A | 77 |
| 09 | HERE: | JMP | HERE | C3 |
| 0A | | | | 09 |
| 0B | | | | 00 |

**Task 2:** Send a logic '0' to the LED (i.e., turn a positive display off or a negative display on).

**Source Program:** (form data initially)

```
        MVI     A,MASKN
        OUT     PORT        ;DATA TO LEDs
        STA     40H         ;SAVE COPY OF OUTPUT DATA
HERE:   JMP     HERE
```

(update data in location 40H)

```
        LXI     H,40H       ;LOAD POINTER TO DATA
        MOV     A,M         ;GET COPY OF OUTPUT DATA
        ANI     MASKN       ;LED BIT = 0
        OUT     PORT        ;DATA TO LEDs
        MOV     M,A         ;SAVE COPY OF OUTPUT DATA
HERE:   JMP     HERE
```

MASKN has a '0' bit in the LED position and 1s elsewhere. Logically ANDing with MASKN does not affect the other bit positions.

**Object Program:** (form data initially)

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | MVI | A,MASKN | 3E |
| 01 | | | | MASKN |
| 02 | | OUT | PORT | D3 |
| 03 | | | | PORT |
| 04 | | STA | 40H | 32 |
| 05 | | | | 40 |
| 06 | | | | 00 |
| 07 | HERE: | JMP | HERE | C3 |
| 08 | | | | 07 |
| 09 | | | | 00 |

**Object Program:** (update data in location 40H)

| | | | | |
|---|---|---|---|---|
| 00 | | LXI | H,40H | 3A |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | MOV | A,M | 7E |
| 04 | | ANI | MASKN | E6 |
| 05 | | | | MASKN |
| 06 | | OUT | PORT | D3 |
| 07 | | | | PORT |
| 08 | | MOV | M,A | 77 |
| 09 | HERE: | JMP | HERE | C3 |
| 0A | | | | 09 |
| 0B | | | | 00 |

## 7-Segment LED Display

**Purpose:** To interface a 7-segment LED display to an 8080 microprocessor. The display may be either common-anode (negative logic) or common-cathode (positive logic).

**COMMON-ANODE AND COMMON-CATHODE DISPLAYS**

Figure 11-13 shows the circuitry required to interface a 7-segment display. Each segment is an individual LED. There are two ways of connecting the LEDs. One is to tie all the cathodes together to ground (see Figure 11-14a); this is a "common-cathode display", and a logic '1' at an anode lights a segment. The other is to tie all the anodes together to a positive voltage supply; this is a "common-anode" display, and a logic '0' at a cathode lights a segment. So, the common-cathode display uses positive logic and the common-anode display uses negative logic. Either display requires appropriate drivers and resistors.

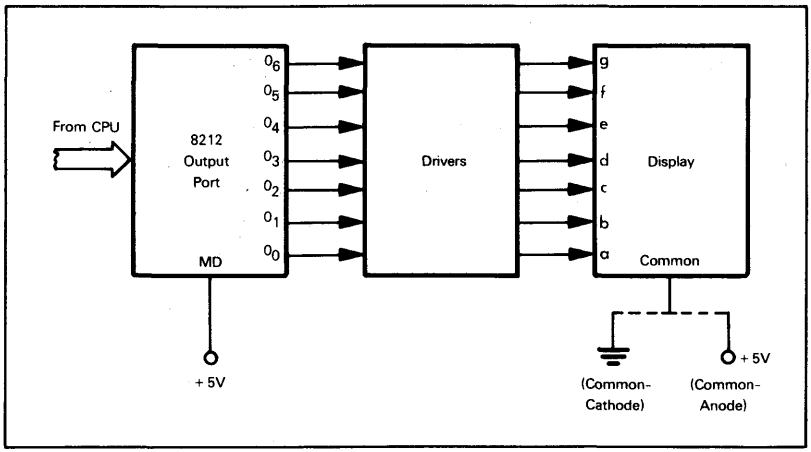
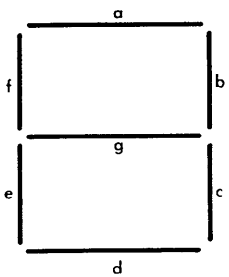



Figure 11-13. Interfacing A 7-Segment Display

The COMMON line from the display is tied either to ground or to +5 volts. The display organization is:




The eighth bit may control a decimal point LED.

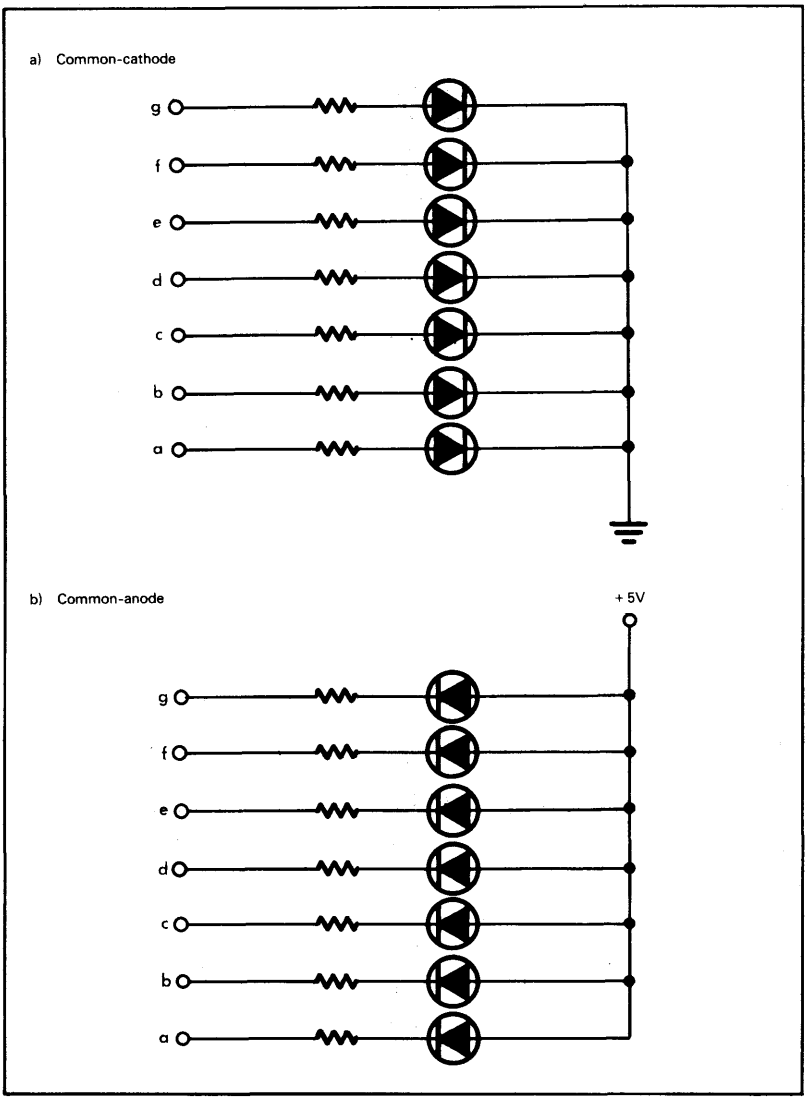



Figure 11-14. Display Organization

**7-SEGMENT REPRESENTATIONS**

Note that the 7-segment display is widely used because it contains the smallest number of separately controlled lights which can provide recognizable representations of all the decimal digits (see Figure 11-15 and Table 11-2). 7-segment displays can also provide some letters and signs (see Tables II-3 and II-4). Better representations require a substantially larger number of display elements and more circuitry. The popularity of 7-segment displays has resulted in the wide availability of low-cost 7-segment decoder/drivers (e.g., 74LS47, 74LS48, 74LS49 and 4511). Some devices also have LAMP TEST inputs (which turn all the display elements on for checking purposes) and blanking inputs and outputs (for blanking leading or trailing zeros).

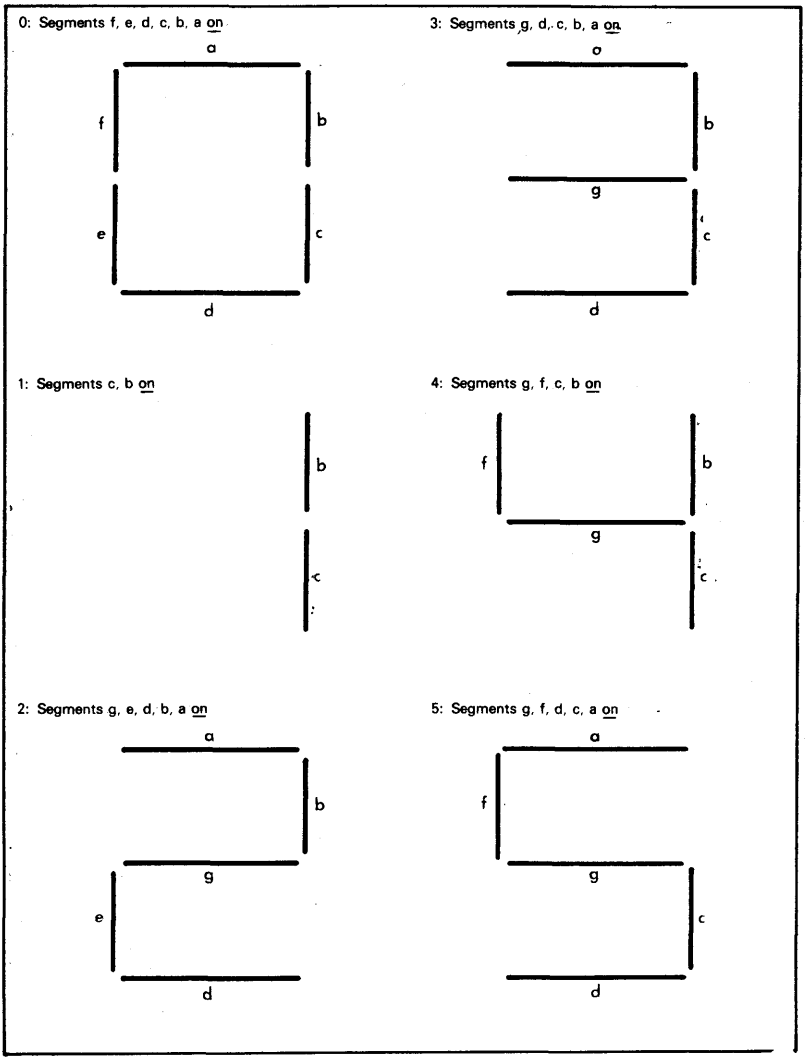



Figure 11-15. 7-Segment Representations Of Decimal Digits

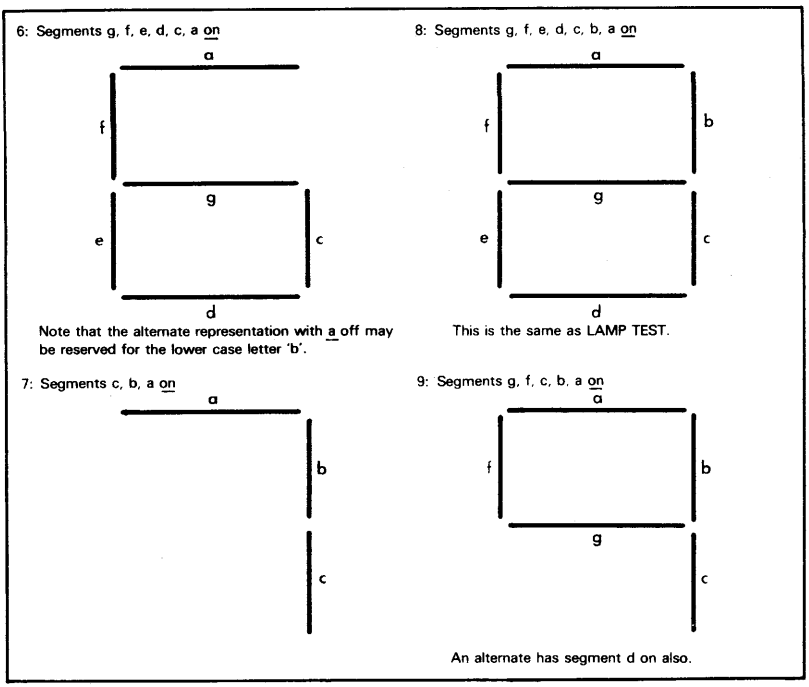



Figure 11-15. 7-Segment Representations Of Decimal Digits (Continued)

Table 11-2. 7-Segment Representation of Decimal Numbers

| Number | Hexadecimal Representation | |
|---|---|---|
| | Common-Cathode | Common-Anode |
| 0 | 3F | 40 |
| 1 | 06 | 79 |
| 2 | 5B | 24 |
| 3 | 4F | 30 |
| 4 | 66 | 19 |
| 5 | 6D | 12 |
| 6 | 7D | 02 |
| 7 | 07 | 78 |
| 8 | 7F | 00 |
| 9 | 67 | 18 |

Bit 7 is always 0, and the others are g, f, e, d, c, b, a in decreasing order of significance.

Table 11-3. 7-Segment Representation Of Letters And Symbols

Upper-Case Letters

| Letter | Hexadecimal Representation | |
|---|---|---|
| | Common-Cathode | Common-Anode |
| A | 77 | 08 |
| C | 39 | 46 |
| E | 79 | 06 |
| F | 71 | 0E |
| H | 76 | 09 |
| I | 06 | 79 |
| J | 1E | 61 |
| L | 38 | 47 |
| O | 3F | 40 |
| P | 73 | 0C |
| U | 3E | 41 |
| Y | 6E | 11 |

Lower-Case Letters and Special Characters

| Letter | Hexadecimal Representation | |
|---|---|---|
| | Common-Cathode | Common-Anode |
| b | 7C | 03 |
| c | 68 | 17 |
| d | 5E | 21 |
| h | 74 | 0B |
| n | 54 | 2B |
| o | 5C | 23 |
| r | 50 | 2F |
| u | 1C | 63 |
| - | 40 | 3F |
| ? | 53 | 2C |

**Programming Example:**

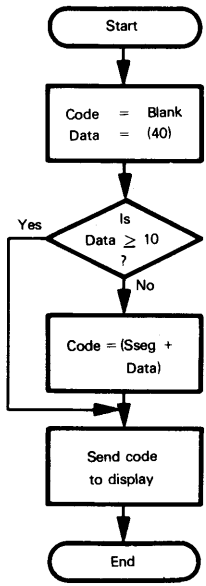
Display the contents of memory location 40 on a 7-segment display if 40 contains a decimal digit. Otherwise, blank the display.

**Sample Problems:**

a. (40) = 05
Result = 5 on display

b. (40) = 66
Result = A blank on display

**Flowchart:**




**Source Program:**

```
        MVI     B,BLANK     GET BLANK CODE
        LDA     40H         GET DATA
        CPI     10          IS DATA > 10?
        JNC     DSPLY       YES, DISPLAY BLANKS
        LXI     D,SSEG      BASE OF 7-SEGMENT TABLE
        MVI     H,0
        MOV     L,A         MAKE DATA INTO 16-BIT INDEX
        DAD     D           INDEX TABLE
        MOV     B,M         GET 7-SEGMENT CODE
DSPLY:  MOV     A,B
        OUT     PORT        CODE TO DISPLAY
HERE:   JMP     HERE
```

BLANK is 00 for a common-cathode display, FF for a common-anode display. An alternative procedure would be to put the blank code at the end of the table and replace all improper data values with 10, i.e.,

```
        LDA     40H         GET DATA
        CPI     10          IS DATA > 10?
        JC      CNVRT       NO, GO CONVERT TO 7-SEGMENT
        MVI     A,10        YES, GET INDEX FOR BLANK CODE
CNVRT:  LXI     D,SSEG      BASE OF 7-SEGMENT TABLE
```

Table SSEG is either the common-cathode or common-anode representation from Table 11-2.

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | MVI | B,BLANK | 06 |
| 01 | | | | BLANK |
| 02 | | LDA | 40H | 3A |
| 03 | | | | 40 |
| 04 | | | | 00 |
| 05 | | CPI | 10 | FE |
| 06 | | | | 0A |
| 07 | | JNC | DSPLY | D2 |
| 08 | | | | 12 |
| 09 | | | | 00 |
| 0A | | LXI | D,SSEG | 11 |
| 0B | | | | 18 |
| 0C | | | | 00 |
| 0D | | MVI | H,0 | 26 |
| 0E | | | | 00 |
| 0F | | MOV | L,A | 6F |
| 10 | | DAD | D | 19 |
| 11 | | MOV | B,M | 46 |
| 12 | | MOV | A,B | 78 |
| 13 | DSPLY: | OUT | PORT | D3 |
| 14 | | | | PORT |
| 15 | HERE: | JMP | HERE | C3 |
| 16 | | | | 15 |
| 17 | | | | 00 |
| 18-21 | | SSEG | | |

## PROBLEMS

### 1) An On-Off Pushbutton

**Purpose:** Each closure of the pushbutton complements (inverts) all the bits in memory location 40. The location initially contains zeros. The program should continuously examine the pushbutton and complement location 40 with each closure.

**Sample Case:**

Location 40 starts with zero.

The first pushbutton closure changes location 40 to FF (hex), the second back to zero, the third back to FF (hex), etc. Assume the pushbutton is debounced in hardware. How would you include debouncing in your program?

### 2) Debouncing a Switch in Software

**Purpose:** Debounce a mechanical switch by waiting until two readings, taken a debounce time apart, give the same result. Assume the debounce time (in ms) is in memory location 40, and place the switch position in memory location 41.

**Sample Problem:**

(40) = 03 causes the program to wait 3 ms between readings.

## 3) Control for a Rotary Switch

**Purpose:** Another switch serves as a LOAD switch for a four-position unencoded rotary switch. The CPU waits for the LOAD switch to close and then reads the position of the rotary switch. This procedure allows the operator to select a final position for a rotary switch before the CPU tries to read it. The program should place the position of the rotary switch in memory location 40. Debounce the LOAD switch in software.

**Sample Problem:**

Place rotary switch in position 2. Close LOAD switch.

Result = (40) = 02

## 4) Record Switch Positions on Lights

**Purpose:** A set of eight switches should have their positions reflected in eight LEDs, i.e., if the switch is closed ('0'), the LED should be off; otherwise, the LED should be on. Assume the CPU output port is connected to the cathodes of the LEDs.

**Sample Problem:**

| | |
|---|---|
| SWITCH 0 | CLOSED |
| SWITCH 1 | OPEN |
| SWITCH 2 | CLOSED |
| SWITCH 3 | OPEN |
| SWITCH 4 | OPEN |
| SWITCH 5 | CLOSED |
| SWITCH 6 | CLOSED |
| SWITCH 7 | OPEN |

Result:

| | |
|---|---|
| LED 0 | OFF |
| LED 1 | ON |
| LED 2 | OFF |
| LED 3 | ON |
| LED 4 | ON |
| LED 5 | OFF |
| LED 6 | OFF |
| LED 7 | ON |

How would you change the program so that a switch attached to bit 7 of Port 2 determined whether the displays were active or not (i.e., if the control switch is closed, the displays reflect the other switches; if the control switch is open, the displays are off)? A control switch is useful when the displays may be distracting to the operator, as in an airplane.

How would you change the program if the control switch were an on-off pushbutton, i.e., each closure reversed the previous state of the displays? Assume that the displays start in the active state and that the program examines and debounces the pushbutton before activating the displays.

## 5) Count on a 7-Segment Display

**Purpose:** The program should count from 0 to 9 continuously on a 7-segment display, starting with zero.

Try different timing lengths for the displays and see what happens. When does the count become visible? What happens if the display is blanked part of the time?

# MORE COMPLEX I/O DEVICES

More complex I/O devices differ from simple keyboards, switches and displays in that:

1) They operate at higher data rates.
2) They may have their own internal clocks and timing.
3) They produce status information and require control information as well as transferring data.

Higher data rates mean that you cannot handle these I/O devices casually. If the processor does not provide the appropriate service, it may miss input data or produce erroneous output data. You are therefore working under much more exacting constraints than in dealing with slower devices. Interrupts are a convenient method for handling complex I/O devices, as we shall see in Chapter 12.

**SYNCHRONIZING WITH I/O DEVICES**

Peripherals like keyboards, teletypewriters, cassettes and floppy disks usually have their own internal timing. These devices generally provide strings or blocks of data, separated by specific timing intervals. The computer must synchronize the initial input or output operation with the peripheral clock and then provide the proper interval between subsequent operations. A simple delay loop like the one shown previously will produce the timing interval. The synchronization may require one or more of the following procedures:

1) Looking for a transition on a clock or strobe line provided by the peripheral for timing purposes. A simple method is to tie the strobe line to the most significant bit of an input port and look for a change in the Sign flag.

2) Finding the center of the time interval during which the data is valid. We would prefer to determine the value of the data at the center of the pulse rather than at the edge where the data may be changing. Finding the center requires a delay of one-half of a transmission interval (bit time) after the edge. Fetching the data at the center also means that small timing errors have little effect on the accuracy of the reception.

3) Recognizing a special starting code. This is easy if the code is a single bit or if we have some timing information. The procedure is more complex if the code is long and could start at any time. Shifting will be necessary to determine where the transmitter is starting its bits, characters, or messages (this is often called a search for the correct "framing").

4) Sampling the data. Taking several samples of the data reduces the probability of receiving it incorrectly from noisy lines. Majority logic can be used to decide on the actual data value.

Reception is, of course, much more difficult than transmission since the peripheral controls the reception and the computer must interpret timing information generated by the peripheral. In transmission, the computer provides the proper timing and formatting for a specific peripheral.

**CONTROL AND STATUS INFORMATION**

Peripherals may require or provide a great deal of information in addition to data and timing. We refer to other information transmitted by the computer as "control information"; it may select modes of operation, start or stop processes, clock registers, enable buffers, choose formats or protocols, provide operator displays, count operations, or identify the type and priority of the operation. We refer to other information transmitted by the peripheral as "status information"; it may indicate the mode of operation, the readiness of devices, the presence of error conditions, the format or protocol in use, and other states or conditions.

The computer handles control and status information just like data. This information often acts like data from a slow peripheral, since it seldom changes even though actual data may be transferred at a high rate. The information may be single bits, digits, words, or multiple words. Single bits or short fields are often combined and handled by a single input or output port.

Combining status and control information into bytes reduces the total number of I/O port addresses required by the peripherals. However, the combination does mean that individual status input bits must be separately interpreted and control output bits must be separately determined. The procedure for isolating status bits is as follows:

**SEPARATING STATUS INFORMATION**

STEP 1) READ STATUS DATA FROM THE PERIPHERAL

STEP 2) LOGICAL AND WITH A MASK (the mask has '1s' in bit positions that must be preserved, '0s' elsewhere).

A shift or flag-setting operation (e.g., ORA A) can replace Step 2 if the field is a single bit and occupies the least significant, most significant, or next to most significant bit position. These positions are often reserved for the most frequently used status information.

**COMBINING CONTROL INFORMATION**

The procedure for setting or resetting control bits is as follows:

STEP 1) LOAD PRIOR CONTROL INFORMATION

STEP 2) LOGICAL AND WITH MASK TO RESET BITS (mask has '0s' in bit positions to be reset, '1s' elsewhere).

STEP 3) LOGICALLY OR WITH MASK TO SET BITS (mask has '1s' in bit positions to be set, '0s' elsewhere).

STEP 4) SEND NEW CONTROL INFORMATION TO PERIPHERAL

Here again the procedure is simpler if the field is a single bit and occupies a position at the end of the word.

This is illustrated by the following examples:

1) A 3-bit field in bit positions 2 through 4 of input port 1 is a scaling factor. Place that factor in the Accumulator.

```
READ STATUS DATA FROM INPUT PORT

        IN      1               ;READ STATUS DATA

MASK OFF SCALING FACTOR AND SHIFT

        ANI     00011100B       ;MASK SCALING FACTOR
        RRC                     ;SHIFT RIGHT TWICE TO NORMALIZE
        RRC
```

2) Bits 2 and 3 in the Accumulator form a 2-bit field which is to be placed into bit positions 5 and 6 of output port 4. Memory location 40 contains bits 0 through 4 and bit 7 of the word to be sent to output port 4, i.e., (40) is the current control information.

```
MOVE DATA TO FIELD POSITIONS

        RLC                     ;SHIFT DATA TO POSITIONS 5 AND 6
        RLC
        RLC
        ANI     01100000B       ;CLEAR OUT OTHER BITS
        MOV     B,A

COMBINE NEW FIELD POSITIONS WITH OLD DATA

        LDA     40H             ;GET OLD DATA
        ANI     10011111B       ;CLEAR FIELD POSITIONS
        ORA     B               ;ADD NEW DATA
        OUT     4
```

A copy of the control information is necessary since the CPU cannot read an output port.

**DOCUMENTING STATUS AND CONTROL TRANSFERS**

Documentation is a serious problem in handling control and status information. The meaning of status inputs or control outputs is seldom obvious. The programmer should clearly indicate the purposes of input and output operations in the comments, e.g., "CHECK IF READER IS ON", "CHOOSE EVEN PARITY OPTION", or "ACTIVATE BIT RATE COUNTER". The Logical and Shift instructions will otherwise be very difficult to remember, understand and debug.

## EXAMPLES

### An Unencoded Keyboard

**Purpose:** Recognize a key closure from an unencoded 3 x 3 keyboard and place the number of the key that was closed in the Accumulator.

Keyboards are just collections of switches (see Figure 11-16). Small numbers of keys are easiest to handle if each key is attached separately to a bit of an input port. Interfacing the keyboard is then the same as interfacing a group of switches.

**MATRIX KEYBOARD**

Keyboards with more than eight keys require more than one input port, and therefore multibyte operations. This is particularly wasteful if the keys are logically separate, i.e., the user will only strike one at a time as in using a calculator or terminal keyboard. The number of input lines required may be reduced by connecting the keys into a matrix as shown in Figure 11-17. Now each key represents a potential connection between a row and a column. The keyboard matrix has n+m external lines, where n is the number of rows and m the number of columns. This compares to n x m external lines if each key is separate. Table 11-4 shows the comparison for some typical configurations.

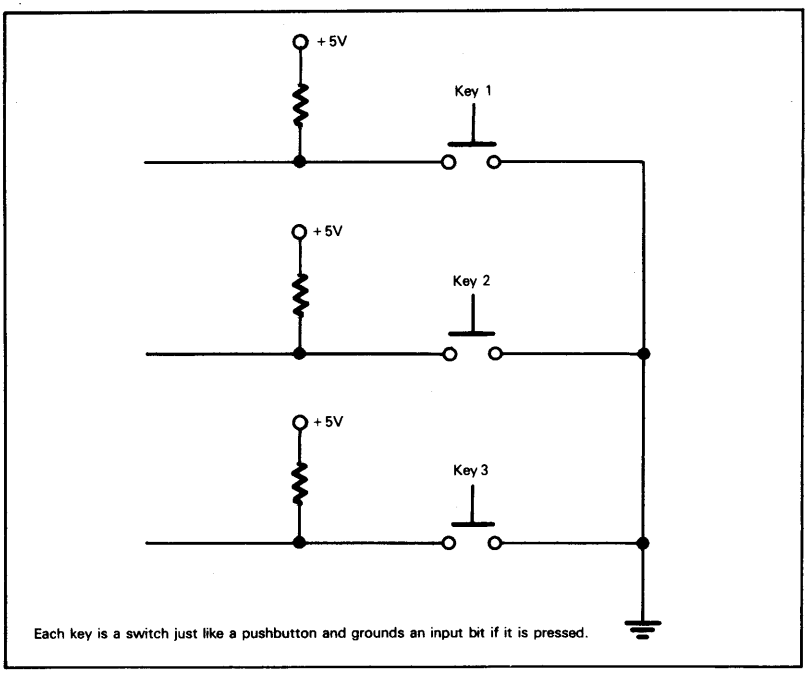



Figure 11-16. A Small Keyboard

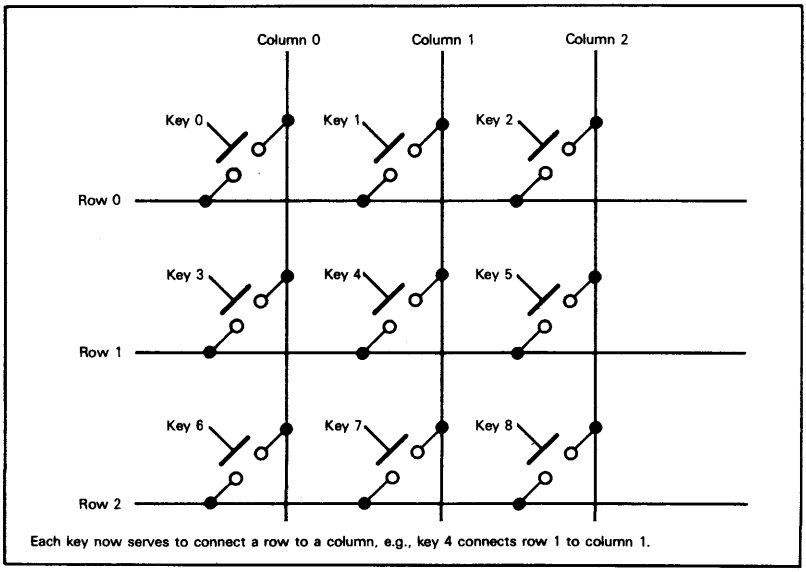



Figure 11-17. A Keyboard Matrix

Table 11-4. Comparison Between Independent Connections And Matrix Connections For Keyboards

| Keyboard Size | Number Of Lines With Independent Connections | Number Of Lines With Matrix Connection |
|---|---|---|
| 3 x 3 | 9 | 6 |
| 4 x 4 | 16 | 8 |
| 4 x 6 | 24 | 10 |
| 5 x 5 | 25 | 10 |
| 6 x 6 | 36 | 12 |
| 6 x 8 | 48 | 14 |
| 8 x 8 | 64 | 16 |

**KEYBOARD SCAN**

The programming involved concerns how to determine which key has been pressed by using the external lines from the matrix. The usual procedure is a "keyboard scan". We ground row 0 and examine the column lines. If any of these lines is grounded, a key in that row has been pressed causing a row-to-column connection. We can determine which key was pressed by determining which column line is grounded, i.e., which bit is '0' at the input port. If none of the column lines is grounded, we proceed to row 1 and repeat the scan of the columns. We can check to see if any keys in the keyboard have been pressed by grounding all the rows at once and examining the columns.

The keyboard scan requires that the row lines be tied to an output port and the column lines to an input port. Figure 11-18 shows the arrangement. The CPU can ground a particular row by placing a '0' in the appropriate bit of the output port and '1s' in the other bits.

The CPU can determine the state of a particular column by examining the appropriate bit of the input port.

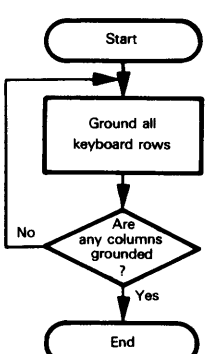
**Task 1:** Wait for a key to be pressed.

**WAITING FOR A KEY CLOSURE**

The procedure is as follows:

1) Ground all the rows by placing '0s' in all the connected bits of the output port.
2) Get column inputs by reading the input port.
3) Return to Step 1 if all the column inputs are '1s'.

**Flowchart:**



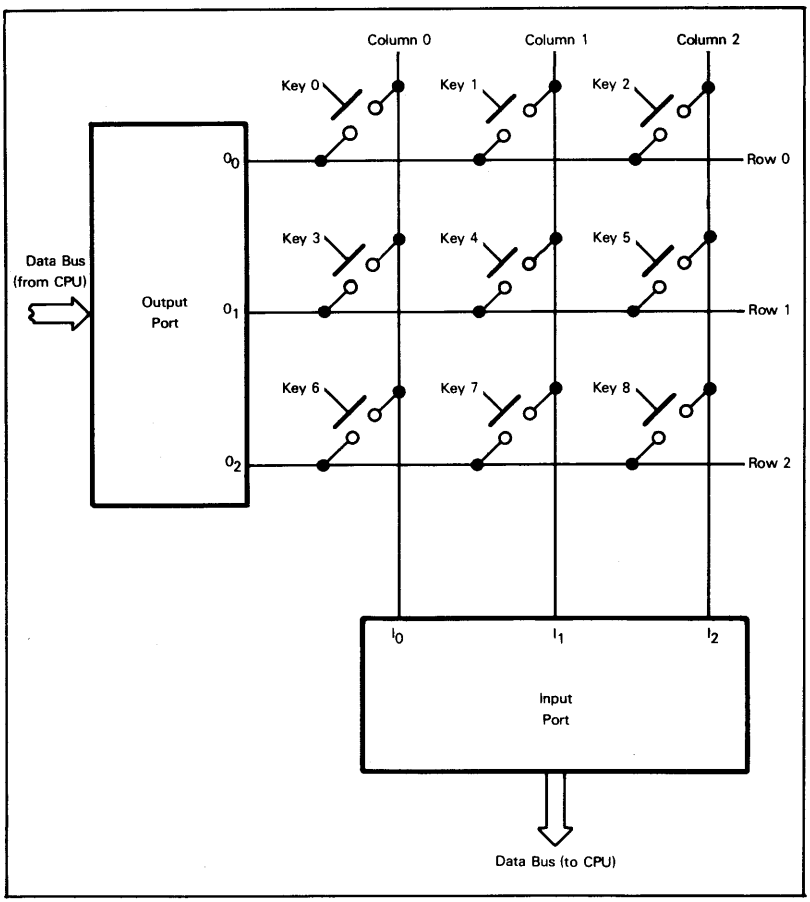



Figure 11-18. I/O Arrangement For A Keyboard Scan

**Source Program:**

```
WAITK:  MVI   A,11111000B
        OUT   KBDOT       ;GROUND ALL KEYBOARD ROWS
        IN    KBDIN       ;GET KEYBOARD COLUMN DATA
        ANI   00000111B   ;MASK COLUMN BITS
        CPI   00000111B   ;ARE ANY COLUMNS GROUNDED?
        JZ    WAITK       ;NO, KEEP LOOKING AT KEYBOARD
DONE:   JMP   DONE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | WAITK: | MVI | A,11111000B | 3E |
| 01 | | | | F8 |
| 02 | | OUT | KBDOT | D3 |
| 03 | | | | KBDOT |
| 04 | | IN | KBDIN | DB |
| 05 | | | | KBDIN |
| 06 | | ANI | 00000111B | E6 |
| 07 | | | | 07 |
| 08 | | CPI | 00000111B | FE |
| 09 | | | | 07 |
| 0A | | JZ | WAITK | CA |
| 0B | | | | 00 |
| 0C | | | | 00 |
| 0D | DONE: | JMP | DONE | C3 |
| 0E | | | | 0D |
| 0F | | | | 00 |

KBDOT is the keyboard output port, KBDIN is the input port.

Masking off the column bits eliminates any problems that could be caused by the states of the unused input lines.

We could generalize the routine by naming the output and masking patterns, i.e.,

```
ALLG    EQU     11111000B
OPEN    EQU     00000111B
```

These names could then be used in the actual program; a different keyboard would only require a change in the definitions and a re-assembly.

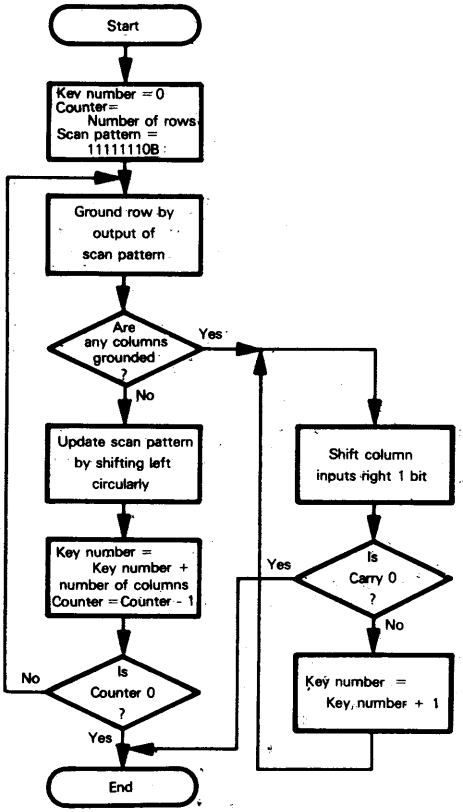
**Task 2:** Identify a key closure by placing the number of the key in the Accumulator.

**IDENTIFYING KEY CLOSURES**

The procedure is as follows:

1) Set key number to 0, counter to number of rows, and output pattern to all '1s' except for a '0' in bit 0.
2) Ground a row by sending the output pattern to the keyboard output port.
3) Update output pattern by shifting the '0' bit left one position.
4) Get column inputs by reading the input port.
5) If any column inputs are '0', proceed to Step 8.
6) Add number of columns to key number to reach next row.
7) Decrement counter. Go to Step 2 if any rows not scanned, otherwise to Step 10.
8) Shift column inputs right one bit.
9) If CARRY=1, add 1 to key number and return to Step 8.
10) End of program.

**Flowchart:**


Start
Key number = 0
Counter =
Number of rows
Scan pattern =
11111110B
Ground row by
output of
scan pattern
Are
any columns
grounded
?
Yes
No
Update scan pattern
by shifting left
circularly
Key number =
Key number +
number of columns
Counter = Counter - 1
Is
Counter 0
?
No
Yes
Shift column
inputs right 1 bit
Is
Carry 0
?
Yes
No
Key number =
Key number + 1
End

**Source Program:**

```
        MVI   B,0           ;KEY NO. = 0
        MVI   C,11111110B   ;START SCAN PATTERN TO GROUND ROW 0
        MVI   D,3           ;COUNTER = NO. OF ROWS
FROW:   MOV   A,C
        OUT   KBDOT         ;SCAN A ROW
        RLC                 ;UPDATE SCAN PATTERN FOR NEXT ROW
        MOV   C,A
        IN    KBDIN         ;GET COLUMN INPUTS
        ANI   00000111B     ;MASK COLUMN BITS
        CPI   00000111B     ;ARE ANY COLUMNS GROUNDED?
        JNZ   FCOL          ;YES, GO FIND WHICH ONE
        MOV   A,B           ;KEY NO = KEY NO.+NO. OF COLUMNS
        ADI   3
        MOV   B,A
        DCR   D             ;HAVE ALL ROWS BEEN SCANNED?
        JNZ   FROW          ;NO, SCAN NEXT ONE
        JMP   DONE          ;YES, DONE
FCOL:   RAR                 ;IS THIS COLUMN GROUNDED?
        JNC   DONE          ;YES, DONE
        INR   B             ;NO, KEY NO. = KEY NO.+1
        JMP   FCOL          ;EXAMINE NEXT COLUMN
DONE:   JMP   DONE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | MVI | B,0 | 06 |
| 01 | | | | 00 |
| 02 | | MVI | C,11111110B | 0E |
| 03 | | | | FE |
| 04 | | MVI | D,3 | 16 |
| 05 | | | | 03 |
| 06 | FROW: | MOV | A,C | 79 |
| 07 | | OUT | KBDOT | D3 |
| 08 | | | | KBDOT |
| 09 | | RLC | | 07 |
| 0A | | MOV | C,A | 4F |
| 0B | | IN | KBDIN | DB |
| 0C | | | | KBDIN |
| 0D | | ANI | 00000111B | E6 |
| 0E | | | | 07 |
| 0F | | CPI | 00000111B | FE |
| 10 | | | | 07 |
| 11 | | JNZ | FCOL | C2 |
| 12 | | | | 1F |
| 13 | | | | 00 |
| 14 | | MOV | A,B | 78 |
| 15 | | ADI | 3 | C6 |
| 16 | | | | 03 |
| 17 | | MOV | B,A | 47 |
| 18 | | DCR | D | 15 |
| 19 | | JNZ | FROW | C2 |
| 1A | | | | 06 |
| 1B | | | | 00 |
| 1C | | JMP | DONE | C3 |
| 1D | | | | 27 |
| 1E | | | | 00 |
| 1F | FCOL: | RAR | | 1F |
| 20 | | JNC | DONE | D2 |
| 21 | | | | 27 |
| 22 | | | | 00 |
| 23 | | INR | B | 04 |
| 24 | | JMP | FCOL | C3 |
| 25 | | | | 1F |
| 26 | | | | 00 |
| 27 | DONE: | JMP | DONE | C3 |
| 28 | | | | 27 |
| 29 | | | | 00 |

Each time a row scan fails, we must add the number of columns to the key number so as to move past the present row (try it on the keyboard in Figure 11-18).

This program can be generalized by making the number of rows, the number of columns, and the masking pattern into named parameters through the use of EQU pseudo-operations.

There are three unconditional Jumps in this program. Can you eliminate two of them by appropriately restructuring the initial conditions? In addition, can you shorten the procedure by which 3 is added to Register B?

## An Encoded Keyboard

**Purpose:** Wait for data to be available from an encoded keyboard which provides a strobe signal along with each data item. Place the data in the Accumulator.

**ENCODED KEYBOARDS**

An encoded keyboard provides a unique identification code for each key. It has internal electronics which perform the scanning and identification procedure described in the previous example. The tradeoff is between the simpler software and higher cost of the encoded keyboard and the more complex software and lower cost of the unencoded keyboard.

Encoded keyboards may use diode matrices, TTL encoders, or MOS encoders. The identification codes may be ASCII, EBCDIC, or some other code. PROMs are often part of the encoding circuitry.

**ROLLOVER**

The encoding circuitry may do more than just transform key closures into codes. It may also debounce the keys and handle "rollover", the problem which arises when more than one key is struck at the same time. Common ways of handling rollover include "2-key rollover" (2KRO), whereby two keys (but not more) struck at the same time are resolved into separate closures, and "n-key rollover" (NKRO), whereby any number of keys struck at the same time are resolved into separate closures.

The encoded keyboard also provides a strobe signal with each new data item. The strobe indicates that new data is present. Figure 11-19 shows the interface between an encoded keyboard and an 8080 microprocessor. The keyboard strobe is latched into the SR flip-flop in the 8212 I/O port (note that the strobe is active-high but the flip-flop output is active-low). A serial input port is also necessary, since the processor has no direct way of determining the status of the SR flip-flop. Output INT from the 8212 is active-low and is intended to serve as an interrupt signal, either directly to the processor or via an 8214 Priority Interrupt Control Unit. Figure 11-20 is a diagram of the 8212 I/O port, and Figure 11-21 describes its use as an input port. (Remember that Volume II of An Introduction To Microcomputers describes the 8212 I/O port in detail.). Most I/O boards used in microcomputer systems contain both parallel data ports and serial status ports.

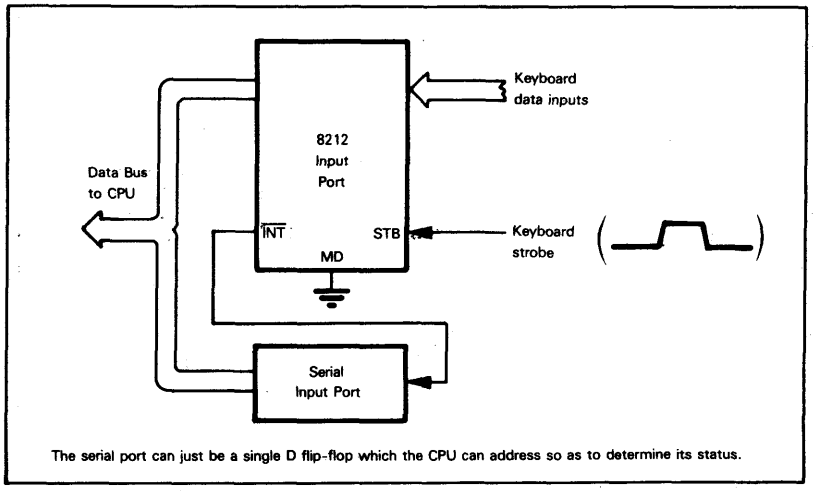
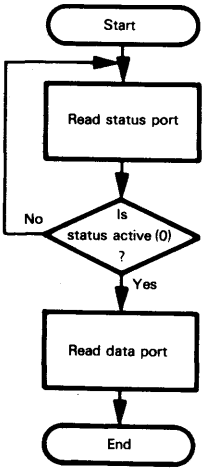



Figure 11-19. I/O Interface For An Encoded Keyboard

**Task:** Wait for an active-low strobe signal at the serial status port and then place the data from the parallel port in the Accumulator.

Note that reading the data from the parallel port clears the SR flip-flop (this circuitry is part of the 8212). We will assume that the serial status port is attached to bit 0 of the Data Bus.

**Flowchart:**




| STB | MD | ($\overline{DS_1}$-$DS_2$) | Data Out Equals |
|---|---|---|---|
| 0 | 0 | 0 | 3 State |
| 1 | 0 | 0 | 3 State |
| 0 | 1 | 0 | Data Latch |
| 1 | 1 | 0 | Data Latch |
| 0 | 0 | 1 | Data Latch |
| 1 | 0 | 1 | Data In |
| 0 | 1 | 1 | Data In |
| 1 | 1 | 1 | Data In |

$\overline{CLR}$ - Resets Data Latch
Sets SR Flip-Flop
(No effect on Output Buffer)

| CLR | ($\overline{DS_1}$-$DS_2$) | STB | * SR | INT |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | ↓ | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | ↓ | 1 | 0 |

* Internal SR Flip-Flop

Figure 11-20. The 8212 I/O Port

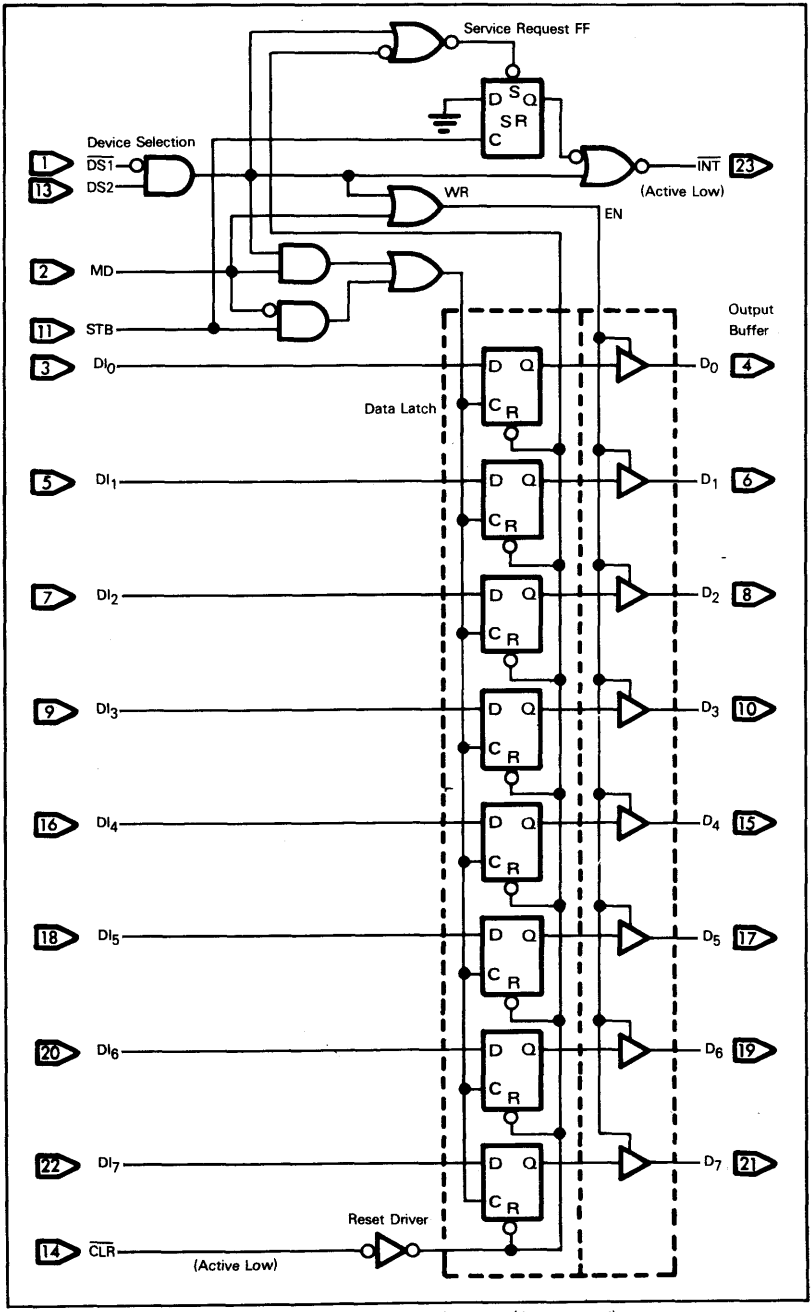



Figure 11-20. The 8212 I/O Port (Continued)

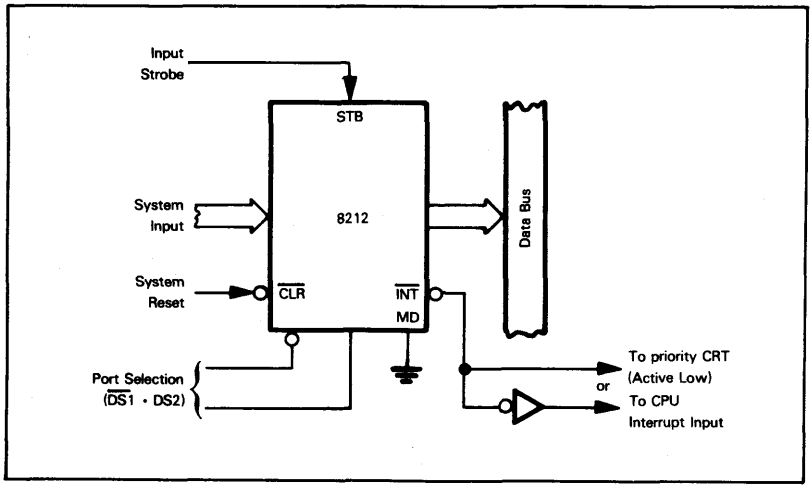



Figure 11-21. The 8212 I/O Port As An Interrupting Input Port

The use of an 8212 as shown in Figure 11-21 is that of a system input port that accepts a strobe from the system input source, which in turn clears the service request flip-flop and interrupts the processor. The processor then goes through a service routine, identifies the port, and causes the device selection logic to go true — enabling the system input data onto the Data Bus.

**Source Program:**

```
SRCHS:  IN      SPORT     ;READ STATUS PORT
        RAR               ;IS THERE NEW KEYBOARD DATA?
        JC      SRCHS     ;NO, KEEP LOOKING
        IN      DPORT     ;YES, FETCH DATA
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | SRCHS: | IN | SPORT | DB |
| 01 | | | | SPORT |
| 02 | | RAR | | 1F |
| 03 | | JC | SRCHS | DA |
| 04 | | | | 00 |
| 05 | | | | 00 |
| 06 | | IN | DPORT | DB |
| 07 | | | | DPORT |
| 08 | HERE: | JMP | HERE | C3 |
| 09 | | | | 08 |
| 0A | | | | 00 |

Remember that the status bit is active-low, i.e., a logic '0' indicates the presence of new data.

## A Digital-To-Analog Converter

**Purpose:** Send data to an 8-bit digital-to-analog converter which requires a LOAD pulse to start the conversion process.

Digital-to-analog converters produce the continuous signals required by motors, solenoids, relays, actuators, and other electrical and mechanical output devices. Typical converters consist of switches and resistor ladders with the appropriate resistance values. Typically, you must provide a reference voltage and some other digital and analog circuitry, although complete units are becoming available at low cost (see, for example, A. Mrozowski, "Analog Output Chips Shrink A-D Conversion Software", Electronics, June 23, 1977, pp. 130-133).

Figure 11-22a describes the 8-bit Signetics NE5018 D/A converter, an example of a device designed to be microprocessor compatible by including an on-chip 8-bit parallel data input latch. A low level on the $\overline{LE}$ (latch enable) input passes the Data Bus information into the latches, where it remains after $\overline{LE}$ goes high. Figure 11-22b illustrates connection of the device to an 8080 system.

Figure 11-23 shows an interface for an 8-bit converter which uses two output ports, one for the converter data and one for the load pulse. The data output port is unnecessary if the converter has a data input register or latch and if the port selection logic can be applied directly to a data strobe input of the converter (as on the Signetics NE5018).

In applications where 8 bits of resolution are not enough, 10- to 16-bit converters can be used. Additional port logic is required to pass all the data bits; some converters provide part of this logic.

Note that the 8212 port is latched by the device selection logic in the output mode (see Figure 11-24). The data therefore remains stable during and after the conversion. The converter typically requires only a few microseconds to produce the desired analog output.

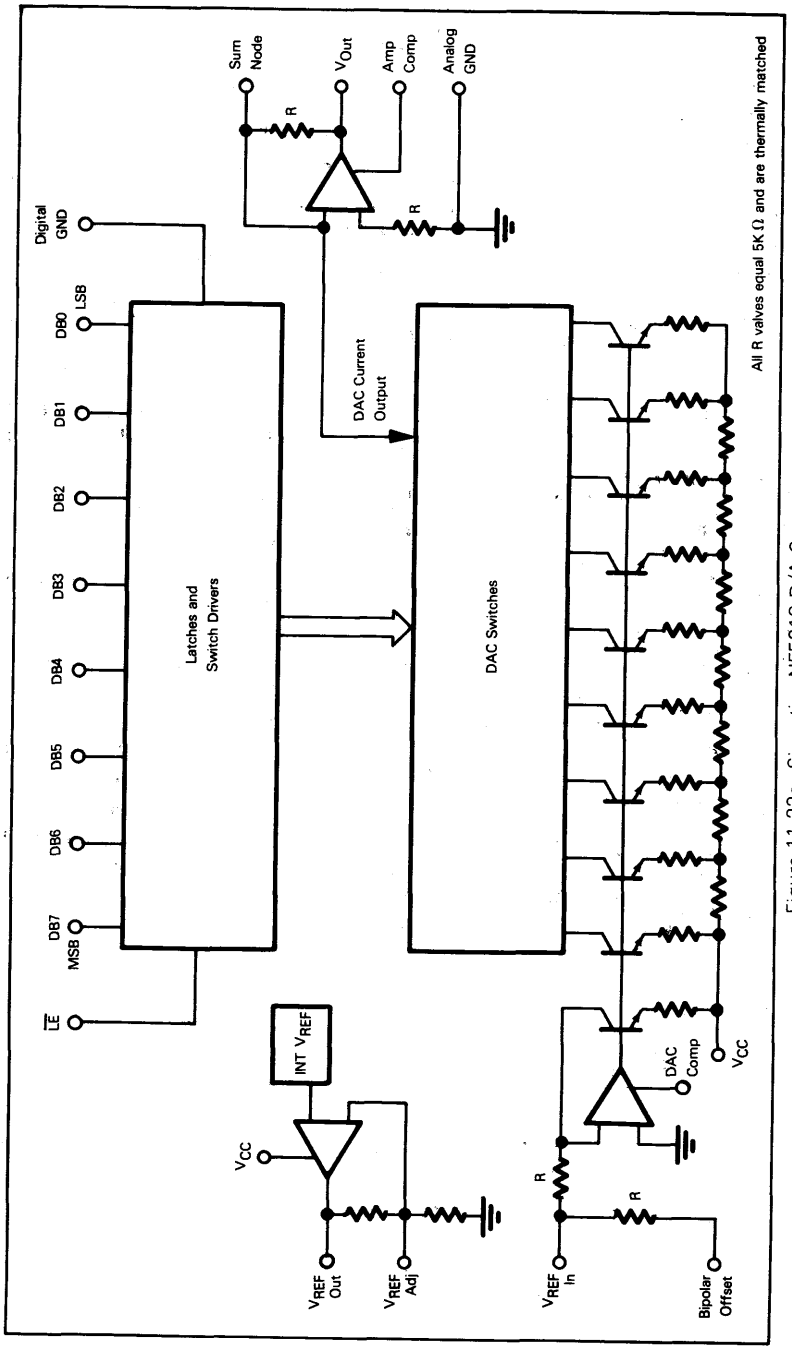



Figure 11-22a. Signetics NE5018 D/A Converter

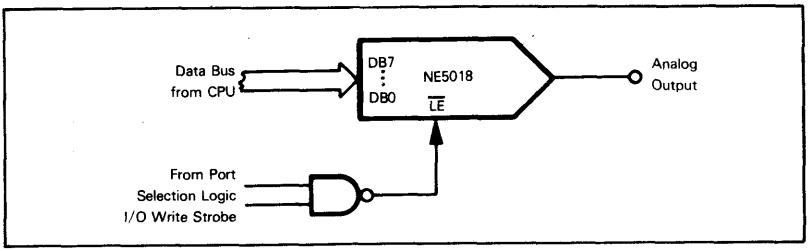



Figure 11-22b. NE5018 Connection To 8080 System

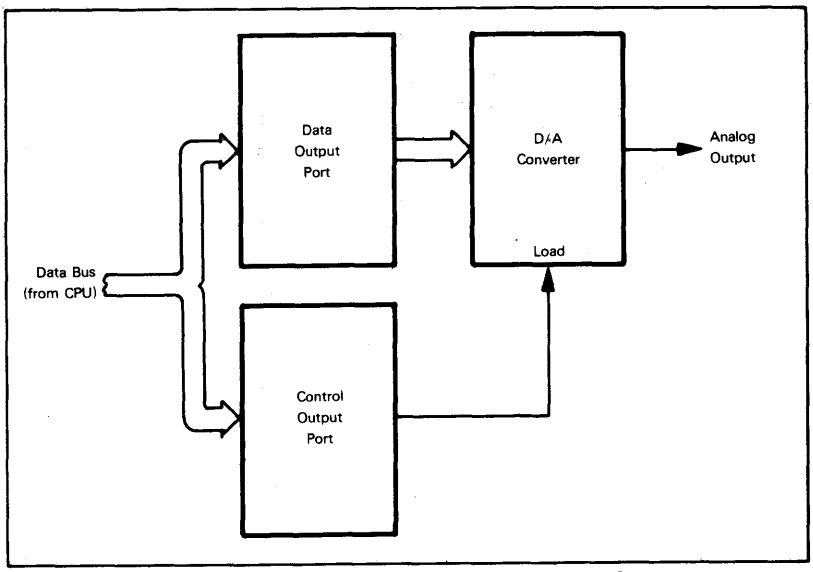



Figure 11-23. Interface For An 8-Bit Digital-To-Analog Converter

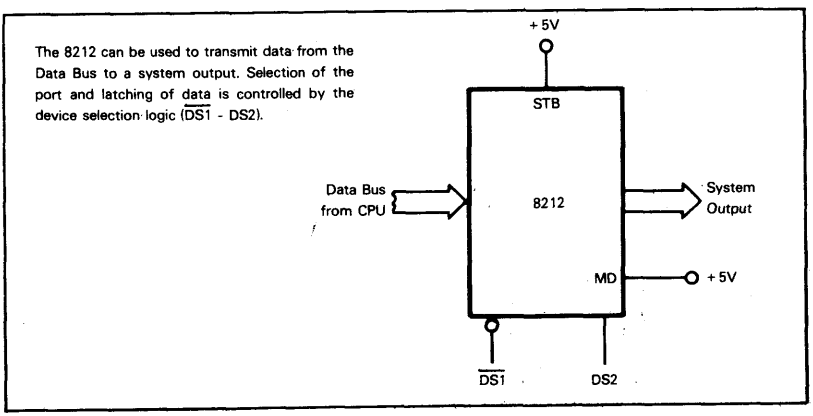
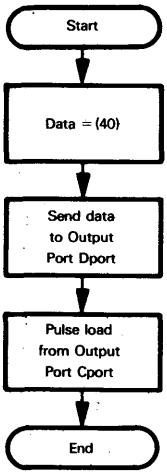



Figure 11-24. The 8212 I/O Port As A Latched Output Port

**Task:** Send the data in memory location 40 to the converter. Send the pulse that will load the converter.

**Flowchart:**




**Source Program:**

(The LOAD input on the DAC is assumed to be attached to Data Bus line 0.)

```
        LDA     40H         ;GET DATA
        OUT     DPORT       ;SEND DATA TO DAC
        MVI     A,1         ;PULSE DAC LOAD
        OUT     CPORT       ;DAC LOAD INPUT HIGH
        SUB     A
        OUT     CPORT       ;DAC LOAD INPUT LOW
HERE:   JMP     HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | Memory Contents (Hex) |
|---|---|---|---|
| 00 | | LDA 40H | 3A |
| 01 | | | 40 |
| 02 | | | 00 |
| 03 | | OUT DPORT | D3 |
| 04 | | | DPORT |
| 05 | | MVI A,1 | 3E |
| 06 | | | 01 |
| 07 | | OUT CPORT | D3 |
| 08 | | | CPORT |
| 09 | | SUB A | 97 |
| 0A | | OUT CPORT | D3 |
| 0B | | | CPORT |
| 0C | HERE: | JMP HERE | C3 |
| 0D | | | 0C |
| 0E | | | 00 |

If the other bits of CPORT are in use, the LOAD signal can be sent high by ORing a copy of the old control data with an appropriate mask. The mask would have a '1' bit in the LOAD position and zeros elsewhere. For example, if the old control data is in memory location 40H, the following instruction sequence would apply:

```
LDA    40H          ;GET OLD CONTROL DATA
ORI    00000001B    ;TURN ON BIT 0
OUT    CPORT
```

The pulse could then be completed by ANDing with a mask which has a '0' bit in the LOAD position and ones elsewhere. For example, if the control data is in the Accumulator,

```
ANI    11111110B
OUT    CPORT
```

would bring the LOAD line low.

The LOAD pulse usually must last a minimum length of time. This could be several clock cycles. When LOAD is low, the analog output remains the same.

The $\overline{\text{INT}}$ signal from the 8212 device can serve as a "BYTE OUT" or other control signal, since the device selection logic activates it. Note, however, that the $\overline{\text{INT}}$ pulse is a fairly brief strobe since it only lasts one clock cycle.

## Analog-To-Digital Converter

**Purpose:** Fetch data from an 8-bit analog-to-digital converter which requires a START CONVERSION pulse to start the conversion process and has a BUSY line to indicate the completion of the process.

Analog-to-digital converters handle the continuous signals produced by various types of sensors and transducers. The converter produces the digital output which the computer requires.

One form of an analog-to-digital converter is the successive-approximation device, which makes a direct 1-bit comparison during each clock cycle. Such converters are fast but have little noise immunity. Dual slope integrating converters are another form of analog-to-digital converters. These devices take longer but are more resistant to noise. Other techniques, such as the incremental charge balancing technique, are also used.

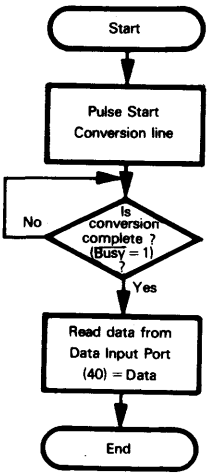
Analog-to-digital converters usually require some external analog and digital circuitry. Complete units are becoming available at low cost.

Figure 11-25 shows the 8-bit Teledyne Semiconductor 8703 A/D converter. The device contains a result latch and three-state data outputs. A pulse on the INITIATE CONVERSION starts conversion of the analog input; after about two milliseconds the result will go to the output latches and the BUSY output will indicate this by switching low. Data is read from the latches by applying a '0' to the $\overline{\text{ENABLE}}$ input.

Figure 11-26 shows the interface for an 8-bit converter. (See also D. Guzeman, "Marry Your $\mu$P to Monolithic A/Ds", Electronic Design, January 18, 1977, pp. 82-86.) The various ports may be unnecessary if the converter (like the Teledyne 8703) has tristate outputs and separate enabling circuitry for the various functions. Note that, besides the input data and status ports, an output control port is necessary to provide the START CONVERSION pulse. The $\overline{\text{BUSY}}$ signal could also latch the converter data into the input data port.

**Task:** Start the conversion process, wait for $\overline{\text{BUSY}}$ to go high, and then read the data and store it in memory location 40.

**Flowchart:**




**Source Program:**

(Both the control and status ports are assumed to be connected to bit 0 of the Data Bus.)

The logical OR and AND operations could be used to pulse the START line.

A delay routine of appropriate length (120 $\mu$s for the AD 7570) could replace the examination of the $\overline{\text{BUSY}}$ signal.

The $\overline{\text{BUSY}}$ signal is assumed to be '0' while the conversion is in progress.

```
        MVI     A,1         ;PULSE START CONVERSION
        OUT     CPORT       ;START CONVERSION = 1
        SUB     A
        OUT     CPORT       ;START CONVERSION = 0
WTBSY:  IN      SPORT       ;IS CONVERSION DONE (BUSY = 1)?
        RAR
        JNC     WTBSY       ;NO, WAIT
        IN      DPORT       ;YES, GET DATA
        STA     40H
HERE:   JMP     HERE
```

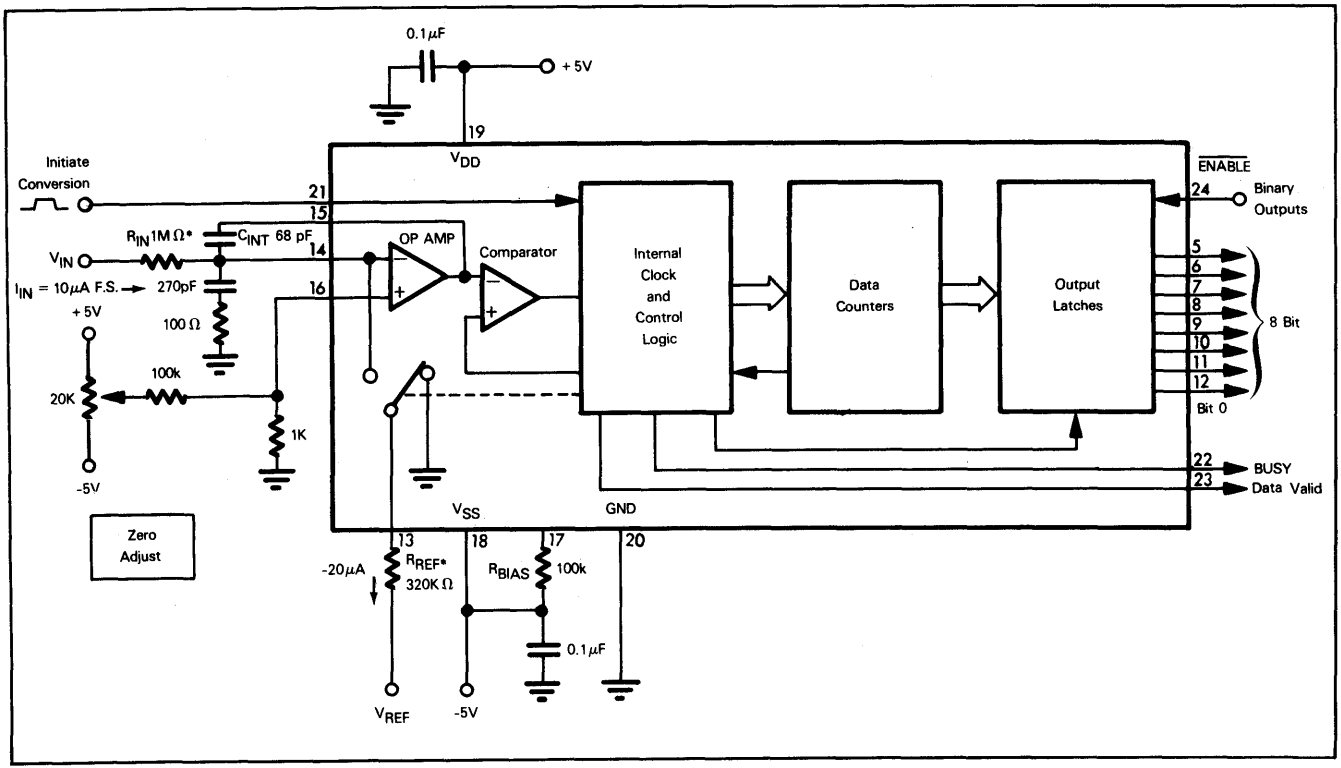



Figure 11-25. Teledyne 8703 A/D Converter

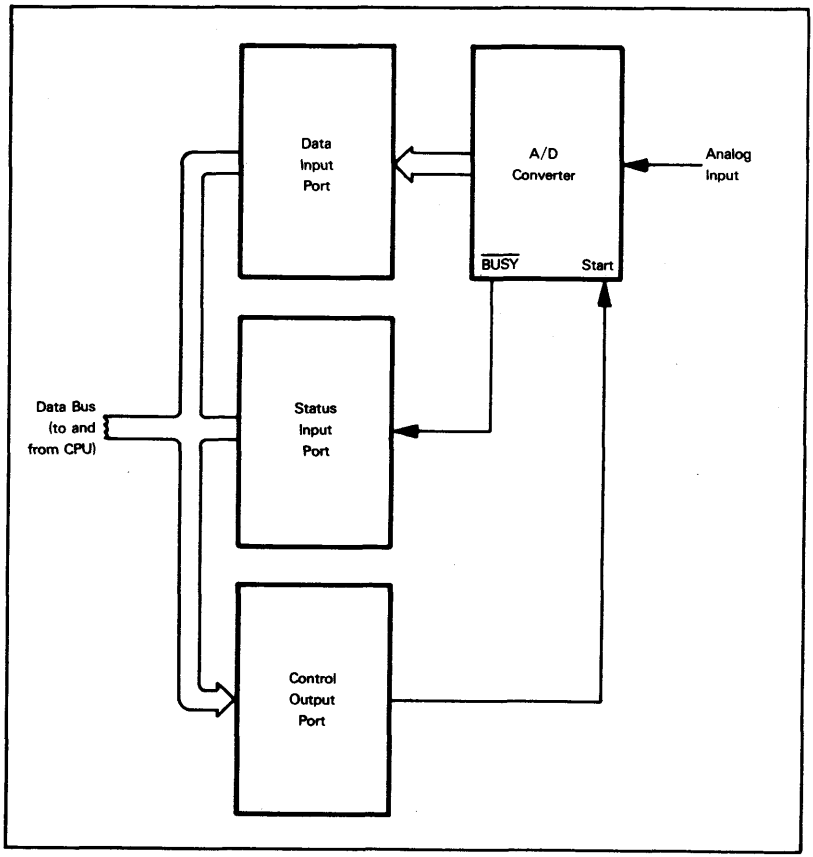



Figure 11-26. Interface For An 8-Bit Analog-To-Digital Converter

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | MVI | A,1 | 3E |
| 01 | | | | 01 |
| 02 | | OUT | CPORT | D3 |
| 03 | | | | CPORT |
| 04 | | SUB | A | 97 |
| 05 | | OUT | CPORT | D3 |
| 06 | | | | CPORT |
| 07 | WTBSY: | IN | SPORT | DB |
| 08 | | | | SPORT |
| 09 | | RAR | | 1F |
| 0A | | JNC | WTBSY | D2 |
| 0B | | | | 07 |
| 0C | | | | 00 |
| 0D | | IN | DPORT | DB |
| 0E | | | | DPORT |
| 0F | | STA | 40H | 32 |
| 10 | | | | 40 |
| 11 | | | | 00 |
| 12 | HERE: | JMP | HERE | C3 |
| 13 | | | | 12 |
| 14 | | | | 00 |

## A Teletypewriter (TTY)

**TTY INTERFACE**

**Purpose:** Transfer data to and from a standard ten-character-per-second serial teletypewriter.

The standard teletypewriter (Teletype Model ASR-33) transfers data in an asynchronous serial mode. The procedure is as follows:

1) The line is normally in the MARK or '1' state.
2) A Start bit (a SPACE or '0' bit) precedes each character.
3) The character is usually 7-bit ASCII with the least significant bit transmitted first.
4) The most significant bit is a Parity bit, which may be even, odd, or fixed at '0' or '1'.
5) Two Stop bits (logic '1') follow each character.

**STANDARD TTY CHARACTER FORMAT**

Figure 11-27 shows the format. Note that each character actually requires the transmission of eleven bits, of which only seven contain information. Since the maximum data rate is ten characters per second, the bit rate is 10 x 11, or 110 Baud. Each bit therefore has a width of 1/110 of a second, or 9.1 milliseconds. This width is an average; the teletypewriter does not maintain it to any high level of accuracy.

For a teletypewriter to communicate properly with a computer, the following procedures are necessary:

RECEIVE (flowcharted in Figure 11-28)

STEP 1) Look for a Start bit, i.e., a logic '0' on the data line.

STEP 2) Center the reception by waiting one-half bit time or 4.55 milliseconds.

STEP 3) Fetch the data bits, waiting one bit time before each one. Enter the data bits into a word by first shifting the bit to the Carry and then circularly shifting the data with the Carry.

STEP 4) Generate the Parity and check it against the received Parity. If they do not match, indicate a "Parity error"

STEP 5) Fetch the Stop bits (waiting one bit time between inputs). If they are not correct (i.e., both Stop bits are not '1'), indicate a "framing error".

**TTY RECEIVE MODE**

**Task:** Fetch data from a teletypewriter through a serial input port and place the data in memory location 60. For procedure see Figure 11-28.

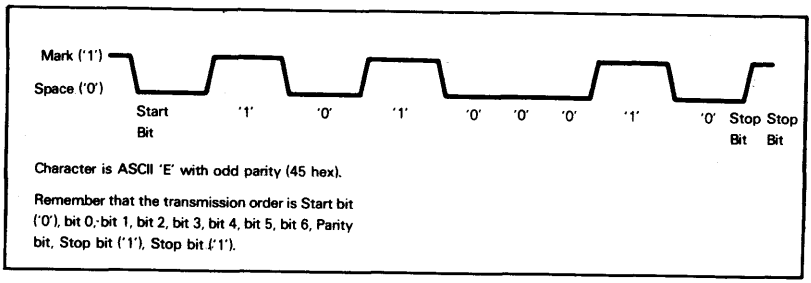



Figure 11-27. Teletypewriter Data Format

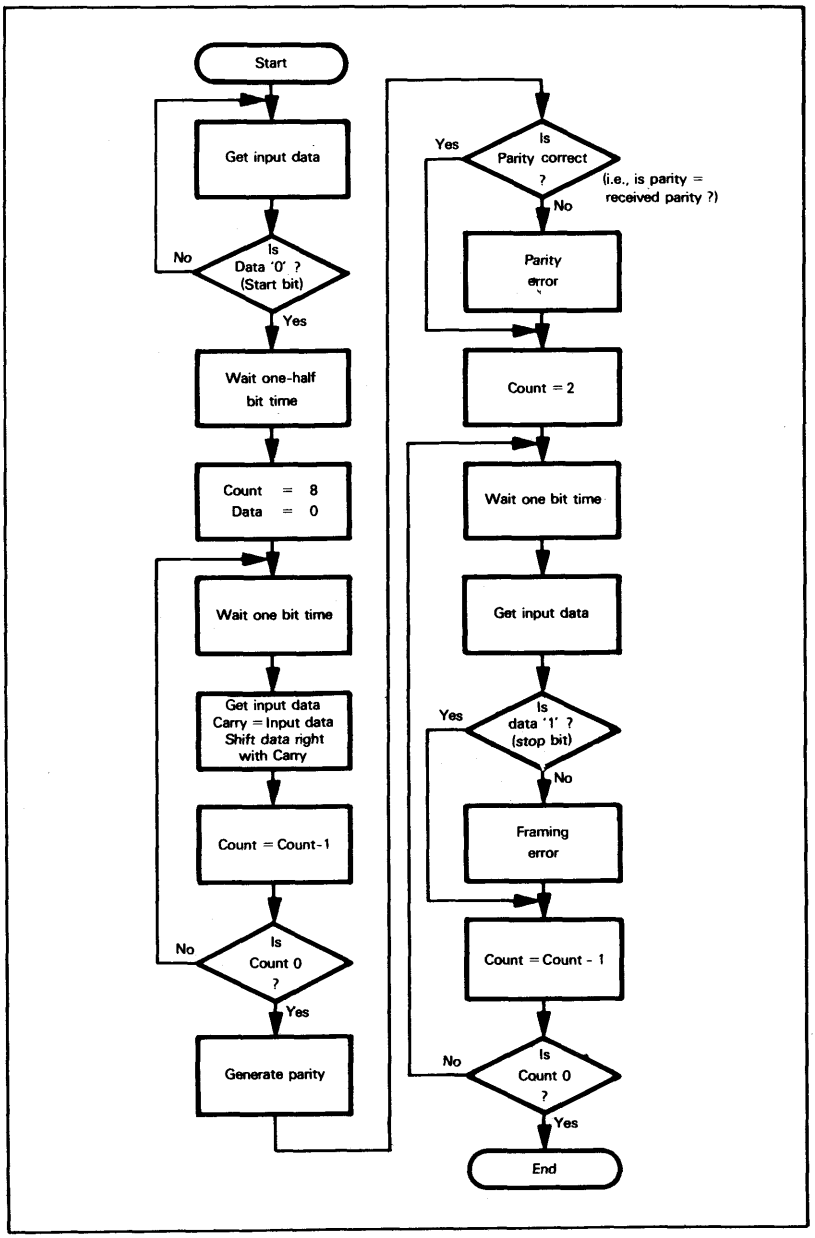



Figure 11-28. Flowchart For Receive Procedure

**Source Program:**

(Assuming serial port is attached to bit 0 of the Data Bus and subroutines PRERR and FRERR handle parity and framing errors respectively.) Parity is odd.

**TTY RECEIVE PROGRAM**

```
;
;WAIT FOR START BIT
;
WTSTB:  IN      RPORT           ;GET DATA
        RAR                     ;IS DATA A START BIT ('0')?
        JC      WTSTB           ;NO, WAIT
;
;CENTER RECEPTION AND MARK DATA WORD
;
        CALL    DHALF           ;WAIT 1/2 BIT TIME
        MVI     D,10000000B     ;MARK DATA WORD
;
;GET SERIAL DATA AND PLACE IN DATA WORD
;
RCVB:   CALL    DFULL           ;WAIT 1 BIT TIME
        IN      RPORT           ;GET SERIAL DATA
        RAR
        MOV     D,A
        RAR                     ;SHIFT SERIAL DATA INTO WORD
        MOV     A,D             ;HAS MARK TRAVERSED WORD?
        JNC     RCVB            ;NO, KEEP FETCHING CHARACTER BITS
;
;CHECK PARITY AND SAVE DATA
;
        ANA     A               ;SET PARITY BIT
        JPE     PRERR           ;PARITY ERROR IF RECEIVED PARITY IS EVEN
        STA     60H
;
;FETCH AND CHECK STOP BITS
;
        MVI     E,2             ;2 STOP BITS
RCVS:   CALL    DFULL           ;WAIT 1 BIT TIME
        IN      RPORT           ;GET SERIAL DATA
        RAR                     ;IS DATA A '1'?
        JNC     FRERR           ;NO, FRAMING ERROR
        DCR     E
        JNZ     RCVS
HERE:   JMP     HERE
;
;DELAY ROUTINES
;
DHALF:  MVI     B,4             ;1/2 BIT TIME
        JMP     DLY
DFULL:  MVI     B,8             ;1 BIT TIME
DLY:    MVI     C,150           ;DELAY 1/8 BIT TIME
        DCR     C
        JNZ     DLY1
        DCR     B
        JNZ     DLY
        RET
```

A simple method for checking to see when a complete word has been received is to place a '1' in the most significant bit of the data word and '0s' elsewhere. When this extra '1' appears in the Carry, a full word of data has been received and shifted in (to the right).

The Stack Pointer must be initialized since the delay routines are called as subroutines.

The delay routine can easily provide both the half bit time delay and the full bit time delay with two separate entry points.

The 8085's serial input line (SID) could also be used. RIM (Read Interrupt Mask) loads the Accumulator with the serial input line in bit 7. So, the required changes are:

```
;
;WAIT FOR START BIT
;
                                    ;
WTSTB:  RIM                         ;GET SERIAL DATA
        RAL                         ;IS SID A START BIT ('0')?
        JC      WTSTB               ;NO, WAIT
;
;GET SERIAL DATA AND PLACE IN DATA WORD
;
RCVB:   CALL    DFULL               ;WAIT 1 BIT TIME
        RIM                         ;GET SERIAL DATA
        RAL
```

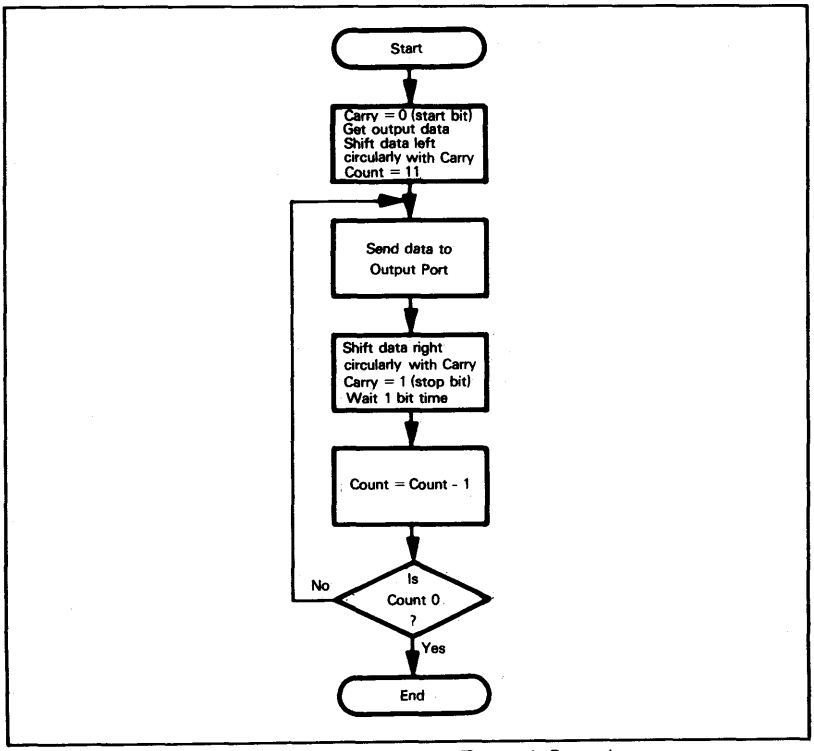



Figure 11-29. Flowchart For Transmit Procedure

TRANSMIT (flowcharted in Figure 11-29)

STEP 1) Transmit a Start bit, i.e., a logic '0'.

STEP 2) Transmit the seven data bits, starting with the least significant bit.

STEP 3) Generate and transmit the Parity bit.

STEP 4) Transmit two Stop bits, i.e., logic '1s'.

**TTY TRANSMIT MODE**

The transmission routine must wait one bit time between each operation.

**UART**

These procedures are sufficiently common and complex to merit a special LSI device, the UART or Universal Asynchronous Receiver/Transmitter. (For a discussion of UARTs, see P. Rony et. al., "The Bugbook IIa", E and L Instruments Inc., 61 First St., Derby, Conn. 06418, or D. G. Larsen et. al., "INWAS: Interfacing with Asynchronous Serial Mode", IEEE Transactions on Industrial Electronics and Control Instrumentation, February 1977, pp. 2-12.) The UART will perform the reception procedure and provide the data in parallel form and a DATA READY signal. It will also accept data in parallel form, perform a transmission procedure, and provide a PERIPHERAL READY signal when it can handle more data. UARTs may have many other features, including:

1) Ability to handle various character lengths (usually five to eight bits), parity options, and numbers of stop bits (usually 1, 1.5, and 2).
2) Indicators for framing errors, parity errors, and "overrun error" (failure to read a character before another one was received).
3) RS-232 (you can find introductory descriptions of RS-232 in F. W. Etcheverry, "Binary Serial Interfaces", EDN, April 20, 1976, pp. 40-43 and in G. Pickles, "Who's Afraid of RS-232?", Kilobaud, May 1977, pp. 50-54) compatibility, i.e., a REQUEST-TO-SEND (RTS) output signal to indicate the presence of data to communications equipment and a CLEAR-TO-SEND (CTS) input signal to indicate, in response to RTS, the readiness of the communications equipment. There may be provisions for other RS-232 signals such as RECEIVED SIGNAL QUALITY, DATA SET READY, or DATA TERMINAL READY.
4) Tristate outputs and control compatibility with a microprocessor.
5) Clock options which allow the UART to sample incoming data several times to detect false start bits and other errors.
6) Interrupt facilities and controls.

**TTY TRANSMIT PROGRAM**

UARTs act as four parallel ports — an input data port, an output data port, an input status port, and an output control port. The status bits include error indicators as well as Ready flags. The control bits select various options. UARTs are inexpensive ($5 to $50, depending on features) and easy to use.

**Task:** Send data from memory location 60 to a teletypewriter through a serial output port.

For procedure see Figure 11-29.

**Source Program:**

(Assuming serial port is attached to bit 0 of the Data Bus and parity is part of data.)

```
GET DATA AND CLEAR START BIT

        LDA     40H         ;GET DATA
        ADD     A           ;CLEAR START BIT
        MVI     D,11        ;COUNT = 11 BITS

TRANSMIT A BIT AND UPDATE DATA

TBIT:   OUT     TPORT       ;TRANSMIT A BIT
        RAR                 ;UPDATE FOR NEXT BIT
        STC                 ;STOP BIT = 1

DELAY 9.1 MS

        MVI     B,8         ;8 TIMES THROUGH
DLY:    MVI     C,150       ;DELAY 1/8 BIT TIME
DLY1:   DCR     C
        JNZ     DLY1
        DCR     B
        JNZ     DLY

COUNT BITS

        DCR     D           ;COUNT DOWN 11 BITS
        JNZ     TBIT
DONE:   JMP     DONE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LDA | 40H | 3A |
| 01 | | | | 40 |
| 02 | | | | 00 |
| 03 | | ADD | A | 87 |
| 04 | | MVI | D,11 | 16 |
| 05 | | | | 0B |
| 06 | TBIT: | OUT | TPORT | D3 |
| 07 | | | | TPORT |
| 08 | | RAR | | 1F |
| 09 | | STC | | 37 |
| 0A | | MVI | B,8 | 06 |
| 0B | | | | 08 |
| 0C | DLY: | MVI | C,150 | 0E |
| 0D | | | | 96 |
| 0E | DLY1: | DCR | C | 0D |
| 0F | | JNZ | DLY1 | C2 |
| 10 | | | | 0E |
| 11 | | | | 00 |
| 12 | | DCR | B | 05 |
| 13 | | JNZ | DLY | C2 |
| 14 | | | | 0C |
| 15 | | | | 00 |
| 16 | | DCR | D | 15 |
| 17 | | JNZ | TBIT | C2 |
| 18 | | | | 06 |
| 19 | | | | 00 |
| 1A | DONE: | JMP | DONE | C3 |
| 1B | | | | 1A |
| 1C | | | | 00 |

The coefficients for the delay routine were calculated from the length of the delay and the execution times of the various instructions at standard 8080 clock rates (i.e., 2 MHz).

MVI - 3.5 $\mu$s
DCR - 2.5 $\mu$s
JNZ - 5.0 $\mu$s

Two registers are necessary, since one cannot provide the full 9.1 milliseconds with the simple routine.

The delay routine can easily produce any multiple of 1/8 of a bit time just by varying the initialization of Register B.

The 8085's serial output line (SOD) could also be used. The SIM instruction loads the SOD output latch with bit 7 of the Accumulator if bit 6 is set. The other bits of the Accumulator have interrupt functions and must have appropriate values (see the 8085 User's Manual or the next chapter).

The new 8085 program is:

```
GET DATA AND CLEAR START BIT

        MVI     D,11        ;COUNT = 11 BITS
        LDA     40H         ;GET DATA
        ANA     A           ;CLEAR START BIT

TRANSMIT A SERIAL BIT THROUGH SOD

TBIT:   MOV     E,A         ;SAVE DATA
        MVI     A,IMASK     ;GET INTERRUPT MASK
        RAR                 ;DATA TO SERIAL OUTPUT BIT (BIT 7)
        SIM                 ;DATA TO SERIAL LATCH

DELAY 9.1 MS

        MVI     B,8         ;8 TIMES THROUGH
DLY:    MVI     C,243       ;DELAY 1/8 BIT TIME
DLY1:   DCR     C
        JNZ     DLY1
        DCR     B
        JNZ     DLY

SHIFT DATA AND COUNT BITS

        STC                 ;STOP BIT = 1
        MOV     A,E         ;SHIFT DATA 1 BIT
        RAL
        DCR     D           ;COUNT DOWN 11 BITS
        JNZ     TBIT
```

IMASK has a '1' in bit 7 to enable loading of the 8085 serial latch (the '1' is in bit 6 after RAL). The coefficients for the delay routine were calculated from the execution times of the various instructions at the standard 8085 clock rate of 3 MHz. The delay routine could preserve Registers B and C with a PUSH B instruction at the start and POP B at the end. High accuracy is unnecessary since the teletypewriter does not maintain high accuracy and each transmission consists of only 11 bits.

ANA A clears the Carry so the start bit is a '0'; STC sets the Carry so the Stop bits are '1s'. No other instructions in the program affect the Carry.

## PROBLEMS

### 1) Separating Closures From An Unencoded Keyboard

**Purpose:** The program should read entries from an unencoded 3 x 3 keyboard and place them in an array. The number of entries required is in memory location 60, and the array starts in memory location 61.

Separate one closure from the next by waiting for the current closure to end. Remember to debounce the keyboard (this can simply be a few millisecond wait).

**Sample Problem:**

(60) = 04

Entries are 7, 2, 2, 4

Result = (61) = 07
(62) = 02
(63) = 02
(64) = 04

### 2) Read A Sentence From An Encoded Keyboard

**Purpose:** The program should read entries from an ASCII keyboard (7 bits with a '0' Parity bit) and place them in an array until it receives an ASCII period (hex 2E). The array starts in memory location 60. Each entry is marked by a strobe, as in the example given under An Encoded Keyboard.

**Sample Problem:**

Entries are H, E, L, L, 0.

Result = (60) = 48 H
(61) = 45 E
(62) = 4C L
(63) = 4C L
(64) = 4F O
(65) = 2E .

### 3) A Variable Amplitude Square Wave Generator

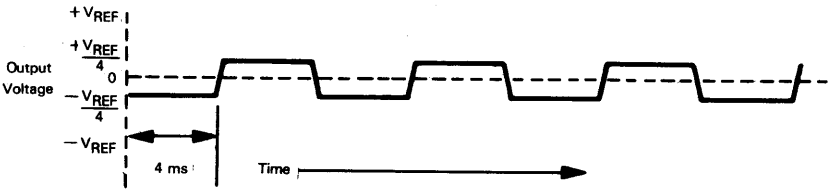
**Purpose:** The program should generate a square wave as shown in the next figure from a D/A converter. Memory location 60 contains the scaled amplitude of the wave, memory location 61 the length of a half-cycle in milliseconds, and memory location 62 the number of cycles.

Assume that a digital output of 80 hex to the converter results in an analog output of 0 volts. For this converter, a digital output of X results in an analog output of $V_{OUT}$ = X-80/80 x-$V_{REF}$ volts.

**Sample Problem:**

(60) = A0 (hex)
(61) = 04
(62) = 03

Result:



The base voltage is 80 hex for 0 volts. Full scale is 00 hex for $-V_{REF}$ volts. So for A0 hex,

$$V_{OUT} = \frac{A0 - 80}{80} \times -V_{REF} = \frac{-V_{REF}}{4}$$

The program produces three pulses of amplitude $V_{REF}/4$ with a half-cycle length of 4 ms.

## 4) Averaging Analog Readings

**Purpose:** The program should take four readings from an A/O converter ten milliseconds apart, and place the average in memory location 60. Assume that the A/D converter takes less than 100 microseconds to convert, so that the conversion time can be ignored.

**Sample Problem:**

Readings are (hex) 86, 89, 81, 84

Result = (60) = 85

## 5) A 30 Characters-Per-Second Terminal

**Purpose:** Modify the transmit and receive routines of the example given under A Teletypewriter to handle a 30 characters-per-second terminal which transfers ASCII data with one stop bit and even parity. How could you write the routines to handle either terminal depending on a flag bit in memory location 40, e.g., (40) = 0 for the 30 characters-per-second terminal, (40) = 1 for the 10 characters-per-second terminal?

# Chapter 12
# INTERRUPTS

Interrupts are serial inputs which the CPU examines as part of each instruction cycle. These inputs allow the CPU to react to events at the hardware level rather than at the software level. Interrupts generally require more hardware than ordinary (programmed) I/O, but provide a faster and more direct response. You may want to review the discussion of interrupts in Volume I of An Introduction to Microcomputers.

**REASONING BEHIND INTERRUPTS**

Why use interrupts? Interrupts allow events such as alarms, power failure, the passage of a certain amount of time, and peripherals having data or being ready to accept data, to get the immediate attention of the CPU. The programmer does not need to have the CPU actually check for the occurrence of these events, nor worry about missing them. An interrupt system is like the bell on a telephone in that it rings when a call is received, so that you go to the telephone in response to the bell rather than having to check the line to see if someone is calling. In the same way, the CPU does not have to wait for an event to occur, but rather responds to the event when it occurs. Of course, this simple description is complicated (just like a telephone system) when there are many interrupts and tasks which cannot be interrupted.

**CHARACTERISTICS OF INTERRUPT SYSTEMS**

The implementation of interrupt systems varies greatly. Among the questions which characterize a particular system are:

1) How many interrupt inputs are there?
2) How many sources of interrupts are there?
3) How does the CPU determine the source of an interrupt if the number of sources exceeds the number of inputs?
4) How does the CPU respond to an interrupt?
5) Can the CPU differentiate between important and unimportant interrupts?
6) How and when is the interrupt system enabled and disabled?

There are many different answers to these questions. The aim of all the implementations, however, is to have the CPU respond rapidly to the interrupt and resume normal activity after the response.

The number of interrupt inputs on the CPU chip determines the number of different responses that the CPU can produce without any additional hardware or software. Each input can produce a different internal response. Unfortunately, most microprocessors have a very small number (one or two, typically) of separate interrupt inputs.

The ultimate response of the CPU to an interrupt must be to transfer control to the correct interrupt service routine and to save the current value of the Program Counter. The CPU must therefore execute a Jump-To-Subroutine or Call instruction with the beginning of the interrupt service routine as its address. This action will save the return address in the Stack and transfer control to the interrupt service routine. The amount of external hardware required to produce this response varies greatly. Some CPUs internally generate the instruction and the address; others require external hardware to form them. The CPU can only generate a different instruction or address for each distinct input.

**POLLING**
**VECTORING**

If the number of interrupting devices exceeds the number of inputs, the CPU will need extra hardware or software to identify the source of the interrupt. In the simplest case, the software can consist of a polling routine which checks the status of the devices which may be interrupting. The only advantage of such a system over normal polling is that the CPU knows that at least one input is active. The alternative solution is for additional hardware to provide a unique data input (or vector) for each source. The two alternatives can be mixed; the vectors can identify groups of inputs which the CPU can differentiate by polling.

**PRIORITY**

An interrupt system which can differentiate between important and unimportant interrupts is called a "priority interrupt system". Internal hardware can provide as many priority levels as there are inputs. External hardware can provide additional levels through the use of a priority register and comparator. The external hardware does not allow the interrupt to get through unless its priority is higher than the contents of the priority register. A priority interrupt system may need a special way to handle low-priority interrupts which may be ignored for long periods of time.

**ENABLING AND DISABLING INTERRUPTS**
**NON-MASKABLE INTERRUPT**

Most interrupt systems can be enabled or disabled. In fact, most CPUs automatically disable interrupts when a RESET is performed (so that the programmer can configure the interrupt system) and on accepting an interrupt (so that the interrupt will not interrupt its own service routine). The programmer may wish to disable interrupts while preparing or processing data, performing a timing loop, or executing a multi-word operation. An interrupt which cannot be disabled (sometimes called a "non-maskable interrupt") may be useful as a power fail interrupt.

**DISADVANTAGES OF INTERRUPTS**

The advantages of interrupts are obvious, but there are also disadvantages. These include:

1) Interrupt systems may require a large amount of extra hardware.
2) Interrupts still require data transfers under program control through the CPU.
3) Interrupts are random inputs which make debugging and testing difficult. Errors may occur sporadically, and therefore may be very hard to find (for a discussion of designing with interrupts, see Baldridge, R.L., "Interrupts Add Power, Complexity to μC-System Design", EDN, August 5, 1977, pp. 67-73).

## 8080 INTERRUPT SYSTEM

The 8080's internal response to an interrupt is very simple. The interrupt system consists of:

1) A single active-high interrupt request input.
2) An interrupt enable flip-flop, the status of which is available as an external output (INTE).
3) A status bit, INTERRUPT ACKNOWLEDGE (INTA), which the CPU places on Data Bus line 0 at the start of each machine cycle.

**8080 INTERRUPT RESPONSE**

The 8080 checks the interrupt line at the end of each instruction cycle. If the interrupt request line is active and interrupts are enabled, the response is as follows (see Volume II of An Introduction to Microcomputers):

1) The CPU disables the interrupt system.
2) The CPU executes an INTERRUPT ACKNOWLEDGE machine cycle. This cycle is distinguished by status bit INTA = 1. DBIN is activated during the cycle so that the CPU will fetch an instruction, but MEMR = 0 so that the CPU will not activate the memory.

External hardware must provide the rest of the response.

### The Restart (RST) Instruction

**RESTART INSTRUCTION**

The 8080 instruction set includes a special instruction, Restart (RST), intended for use with interrupts. Restart is a one-word CALL instruction which saves the current value of the Program Counter in the Stack and jumps to the address specified in the instruction. Table 12-1 contains the various Restart instructions and their destination addresses.

Table 12-1. The Restart Instruction

| Mnemonic Form | Binary Form | Hexadecimal Form | Destination Address (Hex) |
|---|---|---|---|
| RST 0 | 11000111 | C7 | 0000 |
| RST 1 | 11001111 | CF | 0008 |
| RST 2 | 11010111 | D7 | 0010 |
| RST 3 | 11011111 | DF | 0018 |
| RST 4 | 11100111 | E7 | 0020 |
| RST 5 | 11101111 | EF | 0028 |
| RST 6 | 11110111 | F7 | 0030 |
| RST 7 | 11111111 | FF | 0038 |

8085 only (separate inputs)

| | | | |
|---|---|---|---|
| RST 5.5 | | | 002C |
| RST 6.5 | | | 0034 |
| RST 7.5 | | | 003C |

RST is useful in interrupt systems for the following reasons:

1) It is a one-word instruction and so requires only one fetch cycle.
2) It provides eight different destination addresses or vectors.
3) Its vectors are far enough apart to allow Jump instructions to reach the actual service routines.
4) It is easy to form, since five of the bits are always '1'. An 8-to-3 encoder can provide the other three bits quite cheaply.

RST has the following disadvantages:

1) It cannot provide more than eight vectors.
2) Its vectors are not far enough apart to allow space for entire interrupt service routines.
3) Its vectors are in a fixed area of memory.
4) RST 0 has the same destination address as the RESET input and is therefore very difficult to use. The system needs hardware to differentiate between RESET and RST 0, since the two cannot be distinguished by software alone.

**PRODUCING A SINGLE RESTART INSTRUCTION**

The designer can produce a single RST instruction (RST 7) by any of the following methods:

1) Use pullup resistors so that the Data Bus lines are all high when no sources are driving the bus (see Figure 12-1).
2) Tie the $\overline{INTA}$ output of an 8228 System Controller to +12V through a 1K$\Omega$ resistor (see Figure 12-2).
3) Use an 8212 I/O port as an interrupt instruction port (see Figure 12-3) with its inputs tied high and its select line tied to $\overline{INTA}$ (formed by gating INTA and DBIN).

If there is more than one interrupting device, the interrupt service routine starting at memory location 38 hex must identify the source by polling just as with ordinary I/O.

An 8-to-3 encoder can provide all eight RST instructions as shown in Figure 12-4. Remember that the inputs and outputs of a 74LS148 encoder are active-low. As a result, a low level on input $\overline{R}_0$ produces the RST 7 instruction (see Table 12-2), and input $\overline{R}_7$ produces the RST 0 instruction which has the same address as RESET. The encoder only differentiates between simultaneous active inputs, and produces an output which corresponds to the highest priority input.

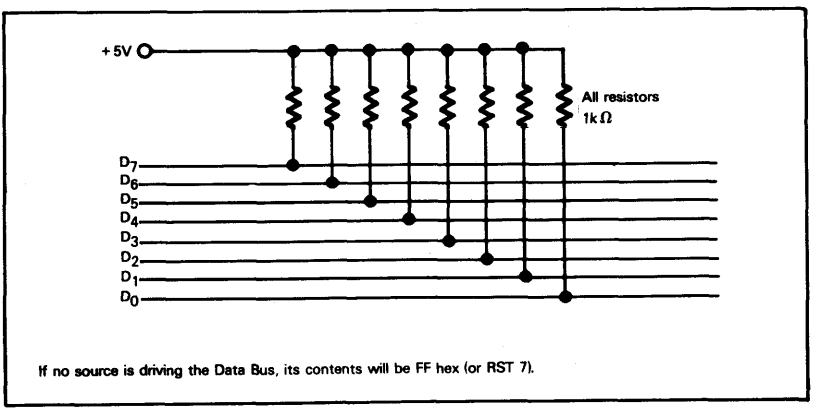
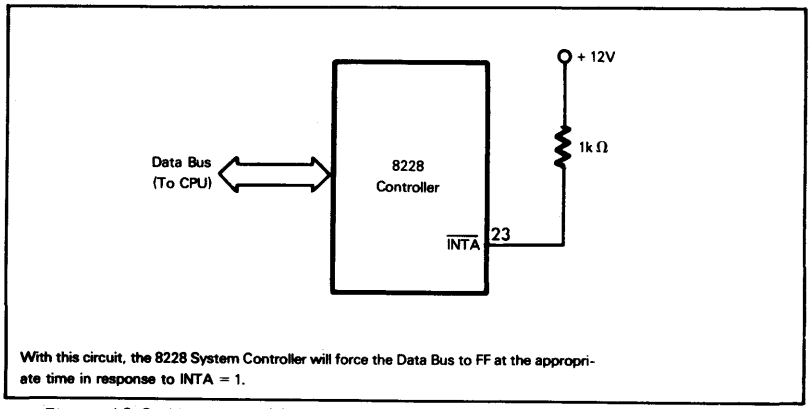
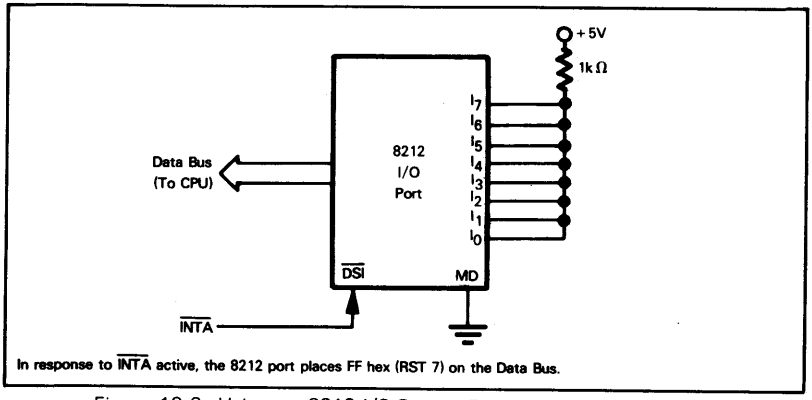



Figure 12-1. Using Pullup Resistors to Form the RST 7 Instruction



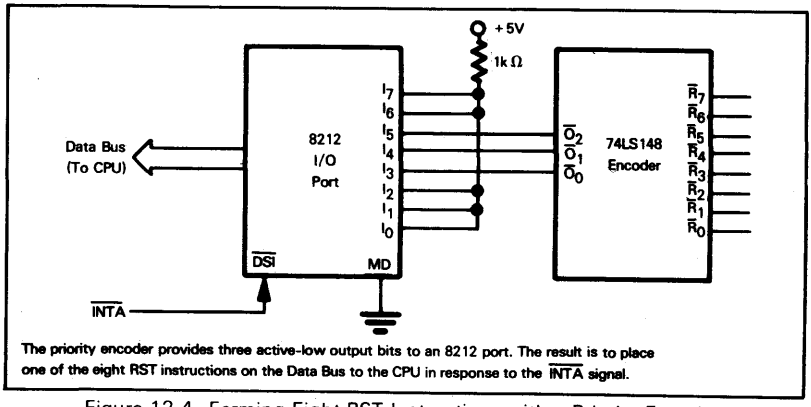

With this circuit, the 8228 System Controller will force the Data Bus to FF at the appropriate time in response to INTA = 1.

Figure 12-2. Using the 8228 System Controller to Form the RST 7 Instruction




In response to INTA active, the 8212 port places FF hex (RST 7) on the Data Bus.

Figure 12-3. Using an 8212 I/O Port to Form the RST 7 Instruction




The priority encoder provides three active-low output bits to an 8212 port. The result is to place one of the eight RST instructions on the Data Bus to the CPU in response to the INTA signal.

Figure 12-4. Forming Eight RST Instructions with a Priority Encoder

## 8085 INTERRUPT SYSTEM

**8085 INTERRUPT SYSTEM**

The 8085 interrupt system differs from the 8080 interrupt system in that the 8085 has four additional interrupt inputs. The four new inputs all take priority over the regular interrupt input (INTR); they are:

1) RST 5.5, which forces a call to address 2C (hex).
2) RST 6.5, which forces a call to address 34 (hex).
3) RST 7.5, which forces a call to address 3C (hex).
4) TRAP (or non-maskable interrupt), which forces a call to address 24 (hex). This input can be used as a power fail interrupt.

The order of priority (highest to lowest) is TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR.

**SIM INSTRUCTION**

RST 5.5, RST 6.5 and RST 7.5 may be masked (disabled) with a SIM instruction. SIM utilizes the bits of the Accumulator as follows:

- Bit 7 = Serial Output Data (SOD)
- Bit 6 = 1 to enable the serial output data, i.e., load the output latch
- Bit 5 is not used
- Bit 4 = 1 to reset the RST 7.5 request
- Bit 3 = 1 to load the interrupt masks from bits 0 through 2
- Bits 0-2 = 0 to enable interrupts RST 5.5 through RST 7.5 respectively

**RIM INSTRUCTION**

Similarly, the RIM instruction can determine the values of the masks and tell if there are any interrupts pending. RIM loads the Accumulator as follows:

- Bit 7 = Serial Input Data (SID)
- Bits 4-6 = 1 for pending requests at levels 5.5, 6.5, and 7.5, respectively
- Bit 3 = 1 to enable the entire interrupt system (except TRAP)
- Bits 0-2 are the current values of the interrupt masks (0 means enabled) for RST 5.5 through RST 7.5, respectively.

## 8214 PRIORITY INTERRUPT CONTROL UNIT

**8214 INTERRUPT CONTROL UNIT**

An 8214 Priority Interrupt Control Unit can provide an almost complete eight-level vectored interrupt system. This device (see Figure 12-5) contains:

1) An 8-to-3 priority encoder.
2) A current Status register.
3) A priority comparator.
4) Various inputs and outputs for control and expansion.

Some of the features of the 8214 device are (see also Volume II of An Introduction To Microcomputers):

1) The outputs are active-low; the RST instructions produced are given by Table 12-2.
2) The current Status register is also active-low, so that 111 is the lowest priority and 000 the highest. Table 12-3 shows which priorities are allowed by various values in the Status register.
3) $\overline{SGS}$ (Status Group Select) is bit 3 of the Status register. If this bit is low, the status comparison is activated and only interrupts above the level given by the Status register will be accepted. If $\overline{SGS}$ is high, the status comparison is bypassed so that interrupts at level 0 will be accepted.
4) The CPU can place a new value in the Status register by addressing it as an output port via $\overline{ECS}$ (Enable Current Status, active-low).
5) Since the Status register is only an output port, the CPU cannot directly determine its contents.

Note that the contents of the 8214 Status register must be managed. You must place an initial value in it (typically $\overline{SGS}$ = 1, e.g., $0F_{16}$) before enabling interrupts. As part of each interrupt service routine, you must place the correct priority value into the Status register before re-enabling interrupts and must restore the old priority value to the Status register before returning to the interrupted program.

Table 12-2. RST Instructions Produced by the Various Request Inputs

| Highest Input Request Active | RST Instruction | Destination Address (hex) |
|---|---|---|
| 0 | RST 7 | 0038 |
| 1 | RST 6 | 0030 |
| 2 | RST 5 | 0028 |
| 3 | RST 4 | 0020 |
| 4 | RST 3 | 0018 |
| 5 | RST 2 | 0010 |
| 6 | RST 1 | 0008 |
| 7 | RST 0 | 0000 |
| Remember that the encoder outputs are active-low. | | |

Table 12-3. Interrupts Allowed For Various Status Register Values

| Status Register Values | | Interrupts Allowed |
|---|---|---|
| $\overline{SGS}$ | B | |
| 0 | 0 | None |
| 0 | 1 | Request 7 |
| 0 | 2 | Request 6 or above |
| 0 | 3 | Request 5 or above |
| 0 | 4 | Request 4 or above |
| 0 | 5 | Request 3 or above |
| 0 | 6 | Request 2 or above |
| 0 | 7 | Request 1 or above |
| 1 | Any | All |

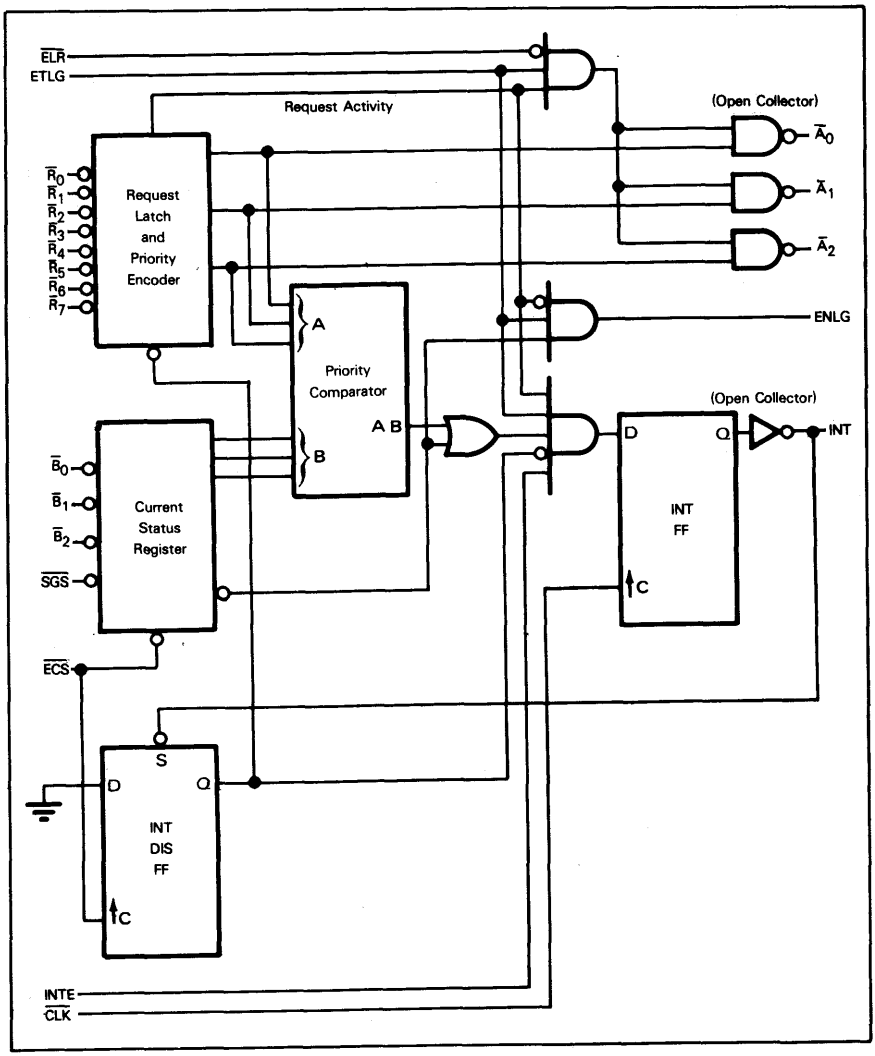



Figure 12-5. The 8214 Priority Interrupt Control Unit

$\overline{SGS}$ must be '1' to enable interrupts at input $\overline{R}_0$, since any priority level only allows interrupts above that level. This procedure is necessary to avoid having a request input interrupt its own service routine.

Figure 12-6 shows the connections required to form an eight-level interrupt controller from an 8214 control unit and an 8212 I/O port.

The advantages of the 8214 device are:

1) It provides eight levels of vectored priority interrupts with minimal hardware.
2) It does not require any timing or other circuitry.
3) It contains the priority register and the comparator.

The disadvantages of the 8214 device are:

1) The priority scheme is inflexible.
2) Low-priority interrupts may be ignored.
3) Expansion of the interrupt system beyond eight levels is difficult.
4) The interrupt vectors are in a fixed area of memory.
5) The CPU cannot read the contents of the priority register. Therefore, the program must retain a copy of the current priority in RAM.

Despite these disadvantages, the 8214 device is a simple, low-cost, complete controller for interrupt systems which do not have a large number of inputs.

## 8259 PROGRAMMABLE INTERRUPT CONTROLLER

**8259 INTERRUPT CONTROLLER**

The 8259 Programmable Interrupt Controller (see Figure 12-7) overcomes some of the disadvantages of the 8214 device. This controller has the following features:

1) It allows several different priority methods, including rotating priority (i.e., the priority levels rotate after each servicing), to insure that all interrupts are eventually serviced. The priority method can be changed under program control.
2) It can place the interrupt vectors anywhere in memory (using either a 32- or 64-byte area).
3) Up to eight of these devices can be combined to provide 64 vectored interrupt levels.

The major disadvantages of the 8259 controller are the programming required and its dependence on the 8228 System Controller to provide the timing pulses which gate a complete Call instruction onto the Data Bus. Volume II of An Introduction To Microcomputers discusses the 8259 device.

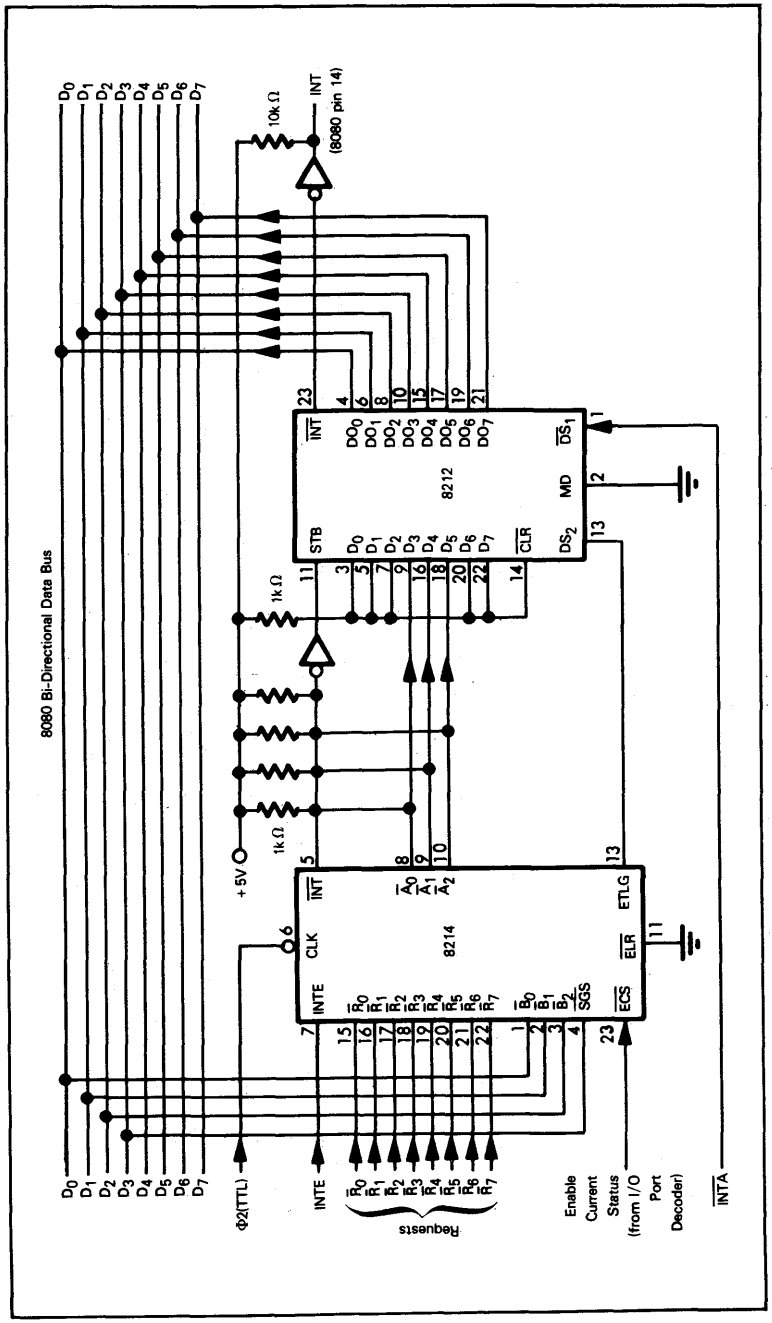



Figure 12-6. The 8214 Priority Interrupt Control Unit Used as an 8-Level Controller

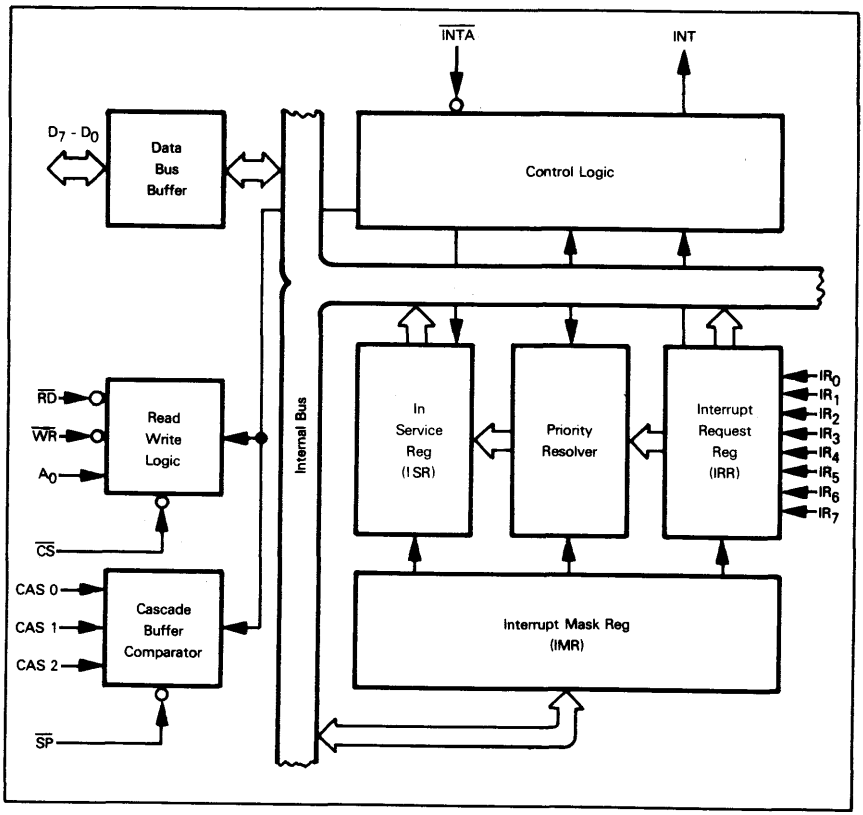



Figure 12-7. The 8259 Programmable Interrupt Controller

## EXAMPLES

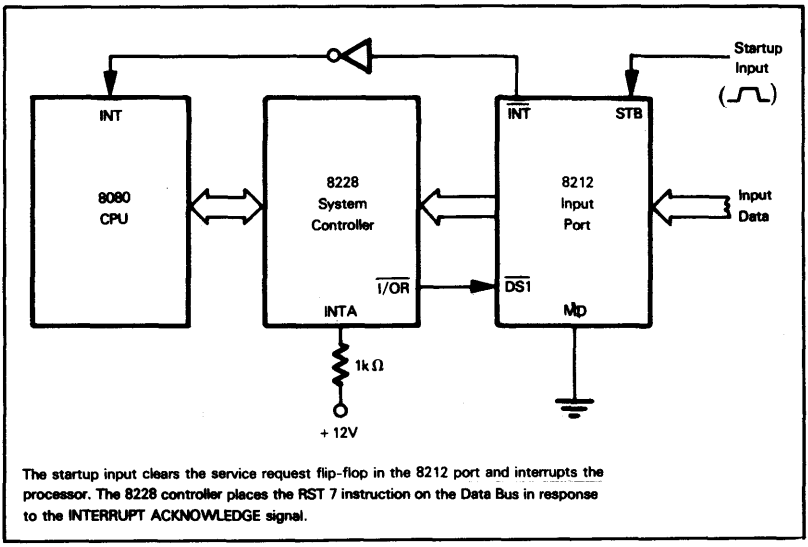
### A Startup Interrupt

**Purpose:** Have the CPU wait until the startup interrupt occurs.

**STARTUP INTERRUPT**

Many systems remain inactive until the operator actually starts them or a DATA READY signal is received. When power-on RESET occurs on such systems, the software must initialize the Stack Pointer, enable interrupts and execute a HLT instruction. (Remember that a RESET disables the interrupts.) In the flowchart, the decision as to whether startup is active is made in hardware (i.e., by the CPU examining the interrupt input internally) rather than in software.

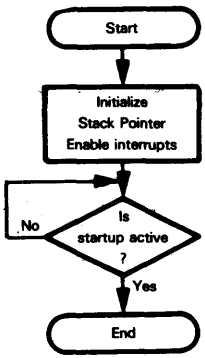
Figure 12-8 shows the circuitry required by the interrupt. Note that the startup input should be active-high since the 8212 service request flip-flop responds to a low-to-high transition on the STB input. Also, the $\overline{INT}$ output from the 8212 must be inverted before being connected to the 8080's active-high interrupt input.




The startup input clears the service request flip-flop in the 8212 port and interrupts the processor. The 8228 controller places the RST 7 instruction on the Data Bus in response to the INTERRUPT ACKNOWLEDGE signal.

Figure 12-8. A Single Input Interrupt

**Flowchart:**




Note that hardware rather than software determines whether the startup is active.

**Source Program:**

```
RESET   EQU   0
        ORG   RESET
        LXI   SP,100H     ;PUT STACK AT END OF MEMORY
        EI                ;ENABLE INTERRUPTS
        HLT               ;AND WAIT
        ORG   INTRP
        IN    PORT        ;CLEAR INTERRUPT
        LXI   SP,100H     ;REINITIALIZE STACK POINTER
HERE:   JMP   HERE
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | | LXI | SP,100H | 31 |
| 01 | | | | 00 |
| 02 | | | | 01 |
| 03 | | EI | | FB |
| 04 | | HLT | | 76 |
| INTRP | | IN | PORT | DB |
| +1 | | | | PORT |
| +2 | | LXI | SP,100H | 31 |
| +3 | | | | 00 |
| +4 | | | | 01 |
| +5 | HERE: | JMP | HERE | C3 |
| +6 | | | | INTRP+5 |
| +7 | | | | |

The program must initialize the Stack Pointer, since the RST instruction uses the Stack. Each iteration reinitializes the Stack Pointer so that the Stack doesn't grow.

The program must initialize the Stack Pointer and the buffer pointer before enabling the interrupts so that the service routine has a place for the data and the return address.

**LOCATION OF INTERRUPT SERVICE ROUTINE**

The exact location of the interrupt service routine varies with the microcomputer. If your microcomputer has no monitor, you can start the interrupt service routine wherever the external hardware directs the CPU. Convenient addresses to use are 38 hexadecimal in 8080 systems (the lowest priority interrupt in 8214-based systems and the only vector in systems without interrupt controllers) or 3C hexadecimal in 8085 systems (the vector for the RST 7.5 input). These addresses are convenient since there are no vectors above them in memory.

**INTERRUPT HANDLING BY MONITORS**

If your microcomputer has a monitor, the monitor will often occupy the RESET and interrupt service addresses. It will then supply service addresses at which you must place either the service routines or the addresses of those routines. A typical monitor routine would be:

```
MONIN:  LHLD    USINT   ;GET USER ADDRESS FOR INTERRUPT
                        ;SERVICE ROUTINE
        PCHL            ;AND JUMP TO IT
```

You must place the address of your service routine in memory locations USINT and USINT+1. Remember that MONIN is an address in the monitor program.

You can include the loading of memory locations USINT and USINT+1 in your main program, i.e.,

```
        LXI     H,INTRP ;GET STARTING ADDRESS OF SERVICE ROUTINE
        SHLD    USINT   ;STORE IT AS USER ADDRESS
```

These instructions come before the enabling of the interrupts.

In this example, the return address which the 8080 stores in the Stack is not useful. However, the main program still must initialize the Stack Pointer so that there is a definite place to put that address. You may not need the LXI SP instruction if the monitor in your microcomputer manages the Stack Pointer.

The service routine discards the return address by simply reinitializing the Stack Pointer. An alternative would be to increment the Stack Pointer twice:

```
INX     SP
INX     SP
```

Remember that accepting an interrupt automatically disables the interrupt system. This allows the real startup routine to configure the interrupt system before re-enabling interrupts. Note that you must deactivate the startup interrupt in the service routine (with the otherwise useless IN PORT instruction) or else it will interrupt again as soon as the interrupt system is re-enabled.

The implementations of the instructions EI (Enable Interrupts) and DI (Disable Interrupts) differ on the 8080. DI takes effect immediately after its execution, while EI takes effect after the execution of the following instruction. The reasoning behind this fact is discussed in Chapter 3 under the description of the EI instruction.

## A Keyboard Interrupt

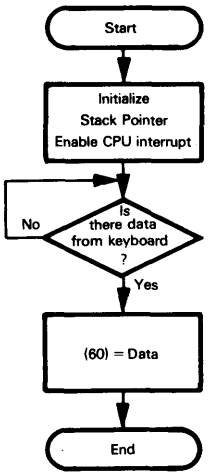
**KEYBOARD INTERRUPT**

**Purpose:** The computer waits for a keyboard interrupt and places the data from the keyboard in memory location 60.

**Sample Problem:**

Keyboard data = 06
Result = (60) = 06

**Flowchart:**



**Source Program:**

```
RESET   EQU   0
        ORG   RESET
        LXI   SP,100H     ;PUT STACK AT END OF MEMORY
        EI                ;ENABLE INTERRUPTS
HERE:   JMP   HERE        ;AND WAIT

        ORG   INTRP
        IN    PORT        ;GET KEYBOARD DATA
        STA   60H         ;SAVE KEYBOARD DATA
        EI                ;RE-ENABLE INTERRUPTS
        RET
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | RESET: | LXI | SP,100H | 31 |
| 01 | | | | 00 |
| 02 | | | | 01 |
| 03 | | EI | | FB |
| 04 | HERE: | JMP | HERE | C3 |
| 05 | | | | 04 |
| 06 | | | | 00 |
| INTRP | | IN | PORT | DB |
| +1 | | | | PORT |
| +2 | | STA | 60H | 32 |
| +3 | | | | 60 |
| +4 | | | | 00 |
| +5 | | EI | | FB |
| +6 | | RET | | C9 |

The JMP HERE is a jump-to-self instruction which is used to represent the main program. After interrupts are enabled in a working system, the main program goes about its business until an interrupt occurs and then resumes execution after the interrupt service routine is completed.

**CHANGING THE RETURN ADDRESS**

The RET instruction at the end of the service routine transfers control back to the JMP instruction. If you want to avoid this, you can simply increment the Program Counter in the Stack, e.g.,

```
POP   H     ;GET RETURN ADDRESS
INX   H     ;INCREMENT RETURN ADDRESS
INX   H
INX   H
PUSH  H     ;PUT IT BACK IN STACK
```

The RET instruction will now transfer control to the instruction following the JMP.

Since the 8080 does not automatically save its registers, you can use them to pass parameters and results between the main program and the interrupt service routine. So, you could leave the data in the Accumulator instead of in memory location 60. This is, however, an extraordinarily dangerous practice which should be avoided in all but the most trivial systems. In general, no interrupt service routine should ever alter any register unless that register's contents have been saved prior to its alteration and will be restored at the completion of the routine.

Note that you must explicitly re-enable the interrupts at the end of the service routine, since the processor disables the interrupt system when it accepts an interrupt. Reading the port deactivates the interrupt.

**FILLING A BUFFER VIA INTERRUPTS**

An alternative would be for the keyboard to keep interrupting until it received an entire sentence, e.g., ending with a carriage return ('CR'). The main program and the service routine would be:

```
RESET:  LXI   SP,100H   ;STACK TO END OF MEMORY
        LXI   H,60H     ;START BUFFER POINTER
        EI              ;ENABLE INTERRUPT
HERE:   JMP   HERE      ;AND WAIT
;
INTRP:  IN    PORT      ;GET KEYBOARD DATA
        MOV   M,A       ;SAVE IN BUFFER
        CPI   CR        ;IS DATA A CARRIAGE RETURN?
        JZ    ENDB      ;YES, END OF SENTENCE
        INX   H         ;NO, INCREMENT BUFFER POINTER
        EI              ;RE-ENABLE INTERRUPTS
        RET
ENDB:   POP   D         ;DISPOSE OF RETURN ADDRESS
        JMP   KBDCODE   ;CONTINUE WITH INTERRUPTS DISABLED
```

**DOUBLE BUFFERING**

When the processor receives a carriage return, it leaves the interrupt system disabled while it handles the sentence. An alternative approach would be to fill another buffer while handling the first one; this approach is called double buffering.

In a real application, the CPU could perform other tasks between interrupts. It could, for instance, edit, move or transmit a line from one buffer while the interrupt was filling another buffer.

## A Printer Interrupt

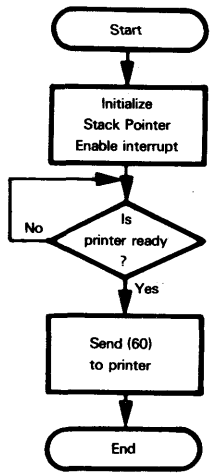
**PRINTER INTERRUPT**

**Purpose:** The computer waits for a printer interrupt and sends the data from memory location 60 to the printer.

**Sample Problem:**

(60) = 41
Result = Printer receives a 41 (ASCII A) when it is ready.

**Flowchart:**




**Source Program:**

```
RESET   EQU   0
        ORG   RESET
        LXI   SP,100H     ;PUT STACK AT END OF MEMORY
        EI                ;ENABLE INTERRUPTS
HERE:   JMP   HERE        ;AND WAIT

        ORG   INTRP
        LDA   60H         ;GET DATA
        OUT   PORT        ;SEND TO PRINTER
        EI                ;RE-ENABLE INTERRUPTS
        RET               ;AND WAIT
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | RESET: | | | |
| | | LXI | SP,100H | 31 |
| 01 | | | | 00 |
| 02 | | | | 01 |
| 03 | | EI | | FB |
| 04 | HERE: | JMP | HERE | C3 |
| 05 | | | | 04 |
| 06 | | | | 00 |
| INTRP | | LDA | 60H | 3A |
| +1 | | | | 60 |
| +2 | | | | 00 |
| +3 | | OUT | PORT | D3 |
| +4 | | | | PORT |
| +5 | | EI | | FB |
| +6 | | RET | | C9 |

Writing data into the interrupting port also deactivates the interrupt.

**EMPTYING A BUFFER WITH INTERRUPTS**

Here again, you could have the printer continue to interrupt until it transferred an entire sentence. The main program and the service routine would be:

```
RESET:  LXI   SP,100H   ;STACK TO END OF MEMORY
        LXI   H,60H     ;START BUFFER POINTER
        EI              ;ENABLE INTERRUPT
HERE:   JMP   HERE      ;AND WAIT
;
INTRP:  MOV   A,M       ;GET CHARACTER FROM
                        ;BUFFER
        OUT   PORT      ;SEND CHARACTER TO PRINTER
        CPI   CR        ;IS CHARACTER A CARRIAGE
                        ;RETURN?
        JZ    ENDB      ;YES, END OF SENTENCE
        INX   H         ;NO, INCREMENT BUFFER
                        ;POINTER
        EI              ;RE-ENABLE INTERRUPTS
        RET
ENDB:   POP   D         ;DISPOSE OF RETURN ADDRESS
        JMP   MAIN      ;CONTINUE WITH INTERRUPTS
                        DISABLED
```

The service routine leaves the interrupts disabled after completing the transfer of the sentence. As before, the CPU could perform other tasks between interrupts.

## A Real-Time Clock Interrupt

**Purpose:** The computer waits for an interrupt from a real-time clock.

**REAL-TIME CLOCK**

A real-time clock simply provides a regular series of pulses which can be used as a time reference. Real-time clock interrupts can be counted to produce any required time interval. A real-time clock can be obtained by dividing down the CPU clock, by using a separate clock generator or a programmable timer like the 8253 device for 8080-based microcomputers, or by using external sources such as the AC line frequency.

**FREQUENCY OF REAL-TIME CLOCK**

Note the tradeoffs involved in determining the frequency of the real-time clock. A high frequency (say 10kHz) allows you to produce a wide range of time intervals to high accuracy. On the other hand, the processor time used in counting clock interrupts may be considerable, and the count will quickly exceed the capacity of a single 8-bit register or memory location. The choice of frequency depends on the precision and timing requirements of your application.

**SYNCHRONIZATION WITH REAL-TIME CLOCK**

One problem is synchronizing processor operations with the real-time clock. Starting at a random point in the clock period will introduce an error into the timing intervals. Some ways to synchronize with the clock are:

1) Start the CPU and clock together. RESET or a startup interrupt can start the clock and the CPU.
2) Allow the CPU to start and stop the clock under program control.
3) Use a high-frequency clock so that an error of one clock period is very small.
4) Line up the CPU and clock by waiting for a clock edge or interrupt before starting the count.

**PRIORITY OF REAL-TIME CLOCK**

A real-time clock interrupt should have very high priority, since the precision of the timing intervals will be affected by any delay in servicing the interrupt. The usual practice is to make the real-time clock the highest priority interrupt except for power failure. The clock interrupt service routine is generally kept extremely short so that it does not interfere with other CPU activities.

### a) Wait For Real-Time Clock

**Source Program:**

```
RESET   EQU   0
        ORG   RESET
        LXI   SP,100H     ;PUT STACK AT END OF MEMORY
        MVI   C,2         ;NUMBER OF INTERRUPTS = 1
        DCR   C           ;SET FLAGS
        EI                ;ENABLE INTERRUPTS
WAITC:  JNZ   WAITC       ;WAIT FOR CLOCK INTERRUPT
HERE:   JMP   HERE
;
        ORG   INTRP
        IN    PORT        ;CLEAR INTERRUPT
        DCR   C           ;COUNT INTERRUPT
        EI                ;RE-ENABLE INTERRUPTS
        RET
```

**Object Program:**

| Memory Address (Hex) | Instruction (Mnemonic) | | | Memory Contents (Hex) |
|---|---|---|---|---|
| 00 | RESET: | | | |
| | | LXI | SP,100H | 31 |
| 01 | | | | 00 |
| 02 | | | | 01 |
| 03 | | MVI | C,2 | 0E |
| 04 | | | | 02 |
| 05 | | DCR | C | 0D |
| 06 | | EI | | FB |
| 07 | WAITC: | | | |
| | | JNZ | WAITC | C2 |
| 08 | | | | 07 |
| 09 | | | | 00 |
| 0A | HERE: | JMP | HERE | C3 |
| 0B | | | | 0A |
| 0C | | | | 00 |
| INTRP | | IN | PORT | DB |
| +1 | | | | PORT |
| +2 | | DCR | C | 0D |
| +3 | | EI | | FB |
| +4 | | RET | | C9 |

The interrupt service routine deactivates the service request flip-flop (with IN PORT), counts the interrupt, and re-enables the interrupt system. Of course, the routine could count up instead of down and could include carries if the counts were too large for a single register. Generally, a few RAM locations are reserved for the real-time clock. Note that A cannot be used for the counter since it is used by the IN PORT instruction which deactivates the interrupt. With the real-time clock, the CPU does not have to execute timing loops or activate a timer. The software is extremely simple and you can obtain any desired timing interval.

**b) Wait For 10 Real-Time Clock Interrupts**

**Source Program:**

```
RESET   EQU   0
        ORG   RESET
        LXI   SP,100H     ;PUT STACK AT END OF MEMORY
        MVI   C,11        ;NUMBER OF INTERRUPTS = 11
        DCR   C           ;SET FLAGS
        EI                ;ENABLE INTERRUPTS
WAITC:  JNZ   WAITC       ;WAIT FOR 10 INTERRUPTS
HERE:   JMP   HERE
;
        ORG   INTRP
        IN    PORT        ;CLEAR INTERRUPT
        DCR   C           ;COUNT INTERRUPT
        EI                ;RE-ENABLE INTERRUPTS
        RET
```

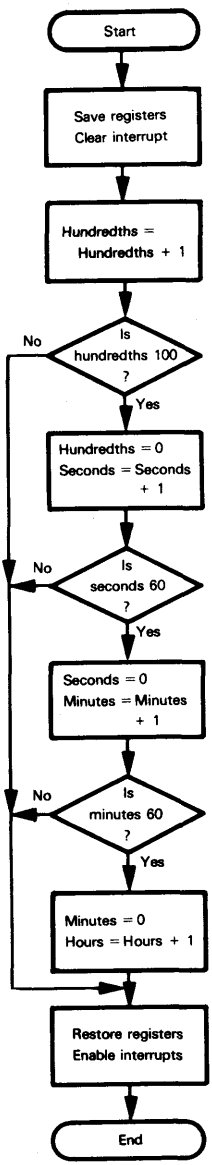
The only change is the new starting value in Register C. Note that you must clear the Zero flag before the CPU executes JNZ WAITC the first time.

**MAINTAINING REAL TIME**

A more general real-time clock interrupt routine may maintain time in several RAM locations. For example, we might use a 100 Hz external clock and keep time as follows in memory starting at address RTCLK:

RTCLK - hundredths of seconds
RTCLK+1 - seconds
RTCLK+2 - minutes
RTCLK+3 - hours

**Flowchart:**


Start
Save registers
Clear interrupt
Hundredths =
Hundredths + 1
Is
hundredths 100
?
No
Yes
Hundredths = 0
Seconds = Seconds
+ 1
Is
seconds 60
?
No
Yes
Seconds = 0
Minutes = Minutes
+ 1
Is
minutes 60
?
No
Yes
Minutes = 0
Hours = Hours + 1
Restore registers
Enable interrupts
End

The interrupt service routine would be:

```
        ORG     INTRP
        PUSH    PSW         ;SAVE REGISTERS
        PUSH    H
        IN      PORT        ;CLEAR REAL-TIME CLOCK INTERRUPT
        LXI     H,RTCLK
        INR     M           ;UPDATE HUNDREDTHS OF SECONDS
        MVI     A,100
        CMP     M           ;IS THERE A CARRY TO SECONDS?
        JNZ     DONE        ;NO, DONE
        MVI     M,0         ;YES, HUNDREDTHS = 0
        INX     H
        INR     M           ;UPDATE SECONDS
        MVI     A,60
        CMP     M           ;IS THERE A CARRY TO MINUTES?
        JNZ     DONE        ;NO, DONE
        MVI     M,0         ;YES, SECONDS = 0.
        INX     H
        INR     M           ;UPDATE MINUTES
        CMP     M           ;IS THERE A CARRY TO HOURS?
        JNZ     DONE        ;NO, DONE
        MVI     M,0         ;YES, MINUTES = 0
        INX     H
        INR     M           ;UPDATE HOURS
DONE:   EI                  ;RE-ENABLE INTERRUPTS
        POP     H           ;RESTORE REGISTERS
        POP     PSW
        RET
```

Now the main program could provide a wait of 300 ms as follows:

```
        LXI     H,RTCLK
        MOV     A,M         ;GET PRESENT TIME (HUNDREDTHS OF SECONDS)
        ADI     30          ;DESIRED TIME IS 30 COUNTS LATER
        CPI     100         ;MOD 100
        JC      WT30
        SUI     100
WT30:   CMP     M           ;WAIT UNTIL DESIRED TIME
        JNZ     WT30
```

Of course, the program could perform other tasks and only check the elapsed time occasionally. How would you produce a delay of seven seconds? Three minutes?

## A Teletypewriter Interrupt

**Purpose:** The processor waits for data to be received from a teletypewriter and stores the data in memory location 60.

**TELETYPEWRITER INTERRUPT**

1) Using a UART

**UART INTERRUPTS**

The receive program is the same as the keyboard interrupt program. The transmit program is the same as the printer interrupt program. An 8080-compatible UART like the 8251 Programmable Communication Interface has TRANSMITTER READY and RECEIVER READY outputs which can be tied to the 8080 or 8085 interrupt inputs. These bits are automatically cleared when the CPU transfers data to and from the UART.

2) Using 8212 Devices

**START BIT INTERRUPT**

An interrupt-driven receive routine must have the received data tied both to an 8212 data input and through an inverter to the 8212 STROBE input. The inverter is necessary since the start bit provides a high-to-low transition while the 8212 only recognizes low-to-high transitions.

**Source Program:**

```
RESET   EQU   0
        ORG   RESET
        LXI   SP,100H     ;PUT STACK AT END OF MEMORY
        EI                ;ENABLE INTERRUPTS
HERE:   JMP   HERE        ;WAIT FOR TTY

        ORG   INTRP
        IN    PORT        ;CLEAR START BIT INTERRUPT
        CALL  TTYRCV      ;GET CHARACTER FROM TTY
        STA   60H         ;SAVE CHARACTER
        EI                ;RE-ENABLE INTERRUPTS
        RET
```

Subroutine TTYRCV is the TTY receive routine shown in the previous chapter without the start bit recognition routine.

The edge used to cause the interrupt is very important here. The transition from the normal MARK or '1' state to the SPACE or '0' state must cause an interrupt, since this transition identifies the start of the character transmission. The '0' to '1' transition will not occur until a non-zero data bit is received.

The service routine must deactivate the start bit interrupt and leave the interrupts disabled until an entire character has been received. Otherwise, each '1' to '0' transition in the character will cause an interrupt. Note that the ignored transitions will still clear the 8212 service request flip-flop. However, reading the data bits from the port will set the flip-flop without causing an interrupt. You must re-enable the interrupts after the entire character has been read.

## MORE GENERAL SERVICE ROUTINES

**TASKS FOR GENERAL SERVICE ROUTINES**

More general service routines that are part of a complete interrupt-driven system must handle the following tasks:

1) Saving all registers in the Stack so that the interrupted program can be correctly resumed.

   Remember that the 8080 Push instruction transfers a register pair to the Stack. PUSH PSW (PSW is the Processor Status Word) transfers the Accumulator and flags to the Stack.

2) Establishing the priority of the interrupt, perhaps by writing the complement of the priority into the 8214 Status register.

   The rest of the interrupt system can then be re-enabled. Remember that in order to restore the old priority correctly you must save it in the Stack as well. A copy of the current priority must be held in RAM, since the 8214 Status register is write-only.

3) Restoring the old priority, all the registers, and the status of the interrupt system before returning to the interrupted program.

The service routine should be transparent as far as the interrupted program is concerned, i.e., it should have no incidental effects.

Any standard subroutines which are used by an interrupt service routine must be re-entrant. If some subroutines cannot be made re-entrant, the interrupt service routine must have separate versions to use.

Priority systems using an 8214 Controller can handle up to eight interrupt sources.

Each interrupt entry point except the one with the lowest priority must contain a jump to the actual service routine, since only a few memory locations are available without running into the next entry point.

**USING RST 0**

If the highest priority interrupt is used, the system must have some way to differentiate this input from RESET. A status flip-flop (port STAT) which holds the highest level interrupt input (i.e., $\overline{INT}$ from the highest priority 8212 port) will do the job. The program is the following:

```
ORG     RESET
JMP     INT0        ;ENTRY FOR RESET OR HIGHEST INTERRUPT
ORG     INT0
STA     TEMP        ;SAVE A IN CASE OF INTERRUPT
IN      STAT        ;IS ENTRY CAUSED BY INTERRUPT?
ORA     A
JNZ     START       ;NO, GO TO RESET ROUTINE
LDA     TEMP        ;YES, RESTORE A
.
.       HIGHEST PRIORITY SERVICE ROUTINE
.
.
.
```

Note that A cannot be saved in the Stack since RESET does not initialize the Stack Pointer. The alternative is to not use the highest interrupt level.

In general, you must be careful when an interrupt may disturb a program. Obviously, timing loops and other routines with precise timing relationships cannot be interrupted. Nor can startup routines or routines that initialize parameters for the service routines. Not only must the service routine save the contents of registers and flags, it must also not interfere with partially completed operations. For example, note that a transfer to or from a buffer with interrupt-driven input/output may not be interrupted. Otherwise, the output routine might be interrupted and have the byte of data it was about to send replaced by a new byte from the input routine. Be particularly cautious when you are handling several words of data. An interrupt could disturb the data and result in part of it being updated and part not. Such mixtures can result in debugging nightmares. More general service routines must save all the registers and the old priority in the Stack on entering, and restore them before returning. Standard procedures are:

**SAVING AND RESTORING REGISTERS AND PRIORITY**

SAVING REGISTERS AND PRIORITY

```
        PUSH    PSW     ;SAVE REGISTERS
        PUSH    B
        PUSH    D
        PUSH    H
        LDA     PRTY    ;SAVE OLD PRIORITY
        PUSH    PSW
        MVI     A,PRTYN ;SET NEW PRIORITY
        STA     PRTY
        OUT     PRREG
        EI
```

RESTORING REGISTERS AND PRIORITY

```
        POP     PSW
        STA     PRTY    ;RESTORE OLD PRIORITY
        OUT     PRREG
        POP     H       ;RESTORE REGISTERS
        POP     D
        POP     B
        POP     PSW
```

If an 8214 device is used, a copy of the old priority is necessary for more than two levels since the 8214 priority register is write-only and the return must restore the correct level.

Of course, the old priority at the lowest level need not be saved or restored since it must have been the lowest possible. Note that the program must write the old priority back into the 8214's Status register before returning.

## PROBLEMS

### 1) A Test Interrupt

**Purpose:** The computer waits for an interrupt to occur and then executes the test instruction:

```
HERE:   JMP   HERE
```

until the next interrupt occurs.

### 2) A Keyboard Interrupt

**Purpose:** The computer waits for a four-digit entry from a keyboard and places the digits in memory locations 60 through 63 (first one received in 60). Each digit entry causes an interrupt. The fourth entry should also result in the disabling of the interrupt system.

**Sample Problem:**

```
Keyboard data = 04, 06, 01, 07
       Result = (60) = 04
                (61) = 06
                (62) = 01
                (63) = 07
```

### 3) A Printer Interrupt

**Purpose:** The computer sends four characters from memory locations 60 to 63 (60 first) to the printer. Each character is requested by an interrupt. The fourth transfer also disables the interrupt system.

### 4) A Real-Time Clock Interrupt

**Purpose:** The computer clears memory location 60 to start and then waits for the real-time clock interrupt. Each time the real-time interrupt occurs, the program complements memory location 60. How would you change the program so that it complements memory location every ten interrupts? How would you change the program so that memory location 60 is zero for ten clock periods, FF (hex) for five clock periods, and so on continuously? You may want to use a display rather than memory location 60 so that it will be easier to see.

### 5) A Teletypewriter Interrupt

**Purpose:** The computer receives TTY data from an interrupting UARTand stores the characters in a buffer starting in memory location 60. When the computer receives a carriage return character (0D hex), it disables the interrupt system and stops receiving characters. How would you change your program to use an 8212 I/O port? Assume that subroutine TTYRCV is available as in the example.

# Chapter 13
# PROBLEM DEFINITION AND PROGRAM DESIGN

## THE TASKS OF SOFTWARE DEVELOPMENT

In the previous chapters, we have concentrated on the writing of short programs in assembly language. While this is an important topic, it is only a small part of the total problem of software development. Although writing assembly language programs seems like a major task to the newcomer, it soon becomes relatively simple. By now, you should be familiar with most of the standard methods for programming in assembly language on the 8080 microprocessor. The next four chapters will describe how to formulate tasks as programs and how to put short programs together to form a working system.

**STAGES OF SOFTWARE DEVELOPMENT**

Software development consists of many stages. Figure 13-1 is a flowchart of the software development process. Its stages are:

- Problem definition
- Program design
- Coding
- Debugging
- Testing
- Documentation
- Maintenance and re-design

Each of these stages is important in the construction of a working system. Note that coding, the writing of programs in a form that the computer understands, is only one of seven stages.

**RELATIVE IMPORTANCE OF CODING**

In fact, coding is usually the easiest stage to define and perform. The rules for writing computer programs are relatively easy to learn. They vary somewhat from computer to computer, but the basic techniques remain the same. Few software projects run into trouble because of the coding stage; indeed, the coding stage is not the most significant part of software development. Experts estimate that a programmer can write one to ten fully debugged and documented statements per day. Clearly, the mere coding of one to ten statements is hardly a full day's effort. On most software projects, coding occupies less than 25% of the programmer's time.

**MEASURING PROGRESS IN STAGES**

Measuring progress in the other stages is difficult. You can say that half of the program has been written, but you can hardly say that half of the errors have been removed or half of the problem has been defined. Clear timetables in such stages as program design, debugging, and testing are difficult to produce. Many days or weeks of effort may result in no clear progress. Furthermore, an incomplete job in one stage may result in tremendous problems later. For example, poor problem definition or program design can make debugging and testing very difficult. Time saved in one stage may be spent many times over in later stages.

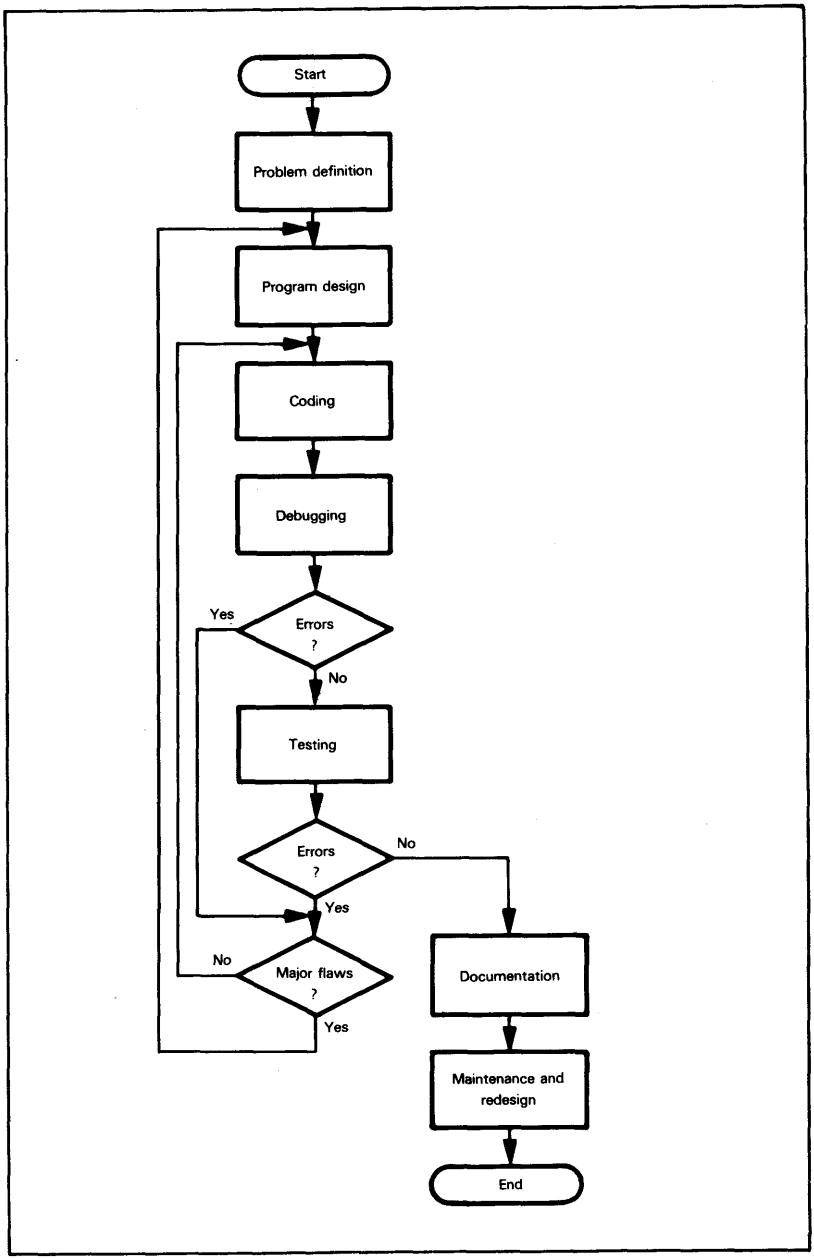



Figure 13-1. Flowchart Of Software Development

## DEFINITION OF THE STAGES

**PROBLEM DEFINITION**

Problem definition is the formulation of the task in terms of the requirements it places on the computer. For example, what is necessary to make a computer control a tool, run a series of electrical tests or handle communications between a central controller and a remote instrument? Problem definition requires that you determine the form and rate of inputs and outputs, the amount and speed of processing that is needed, and the types of possible errors and their handling. Problem definition takes a vaguely defined idea of building a computer-controlled system and defines the tasks and requirements for the computer.

**PROGRAM DESIGN**

Program design is the outline of the computer program which will perform the tasks that have been defined. In the design the tasks are described in a way that can easily be converted into a program. Among the useful techniques in this stage are flowcharting, structured programming, modular programming and top-down design.

**CODING**

Coding is the writing of the program in a form that the computer can either directly understand or translate. The form may be machine language, assembly language or a high-level language.

**DEBUGGING**

Debugging, also called program verification, is making the program do what the design specified that it would do. In this stage, you use such tools as breakpoints, traces, simulators, logic analyzers and in-circuit emulators. The end of the debugging stage is hard to define, since you never know when you have found the last error.

**TESTING**

Testing, also referred to as program validation, is ensuring that the program performs the overall system tasks correctly. The designer uses simulators, exercisers and various statistical techniques to get some measure of the program's performance.

**DOCUMENTATION**

Documentation is the description of the program in the proper form for users and maintenance personnel. Documentation also allows the designer to develop a program library so that subsequent tasks will be far simpler. Flowcharts, comments, memory maps and library forms are some of the tools used in documentation.

**MAINTENANCE AND RE-DESIGN**

Maintenance and re-design are the servicing, improvement and extension of the program. Clearly, the designer must be ready to handle field problems in computer-based equipment. Special diagnostic modes or programs and other maintenance tools may be required. Upgrading or extension of the program may be necessary to make the program meet new requirements or handle new tasks.

The rest of this chapter will consider only the problem definition and program design stages. Chapter 14 will discuss debugging and testing, and Chapter 15 will discuss documentation, extension and re-design. We will bring all the stages together in some simple systems examples in Chapter 16.

## PROBLEM DEFINITION

Typical microprocessor tasks require a lot of definition. For example, what must a program do to control a scale, a cash register or a signal generator? Clearly, we have a long way to go just to define the tasks involved.

## DEFINING THE INPUTS

How do we start the definition? The obvious place to begin is with the inputs. We should begin by listing all the inputs that the computer may receive in this application.

Examples of inputs are:

- Data blocks from transmission lines
- Status words from peripherals
- Data from A/D converters

Then, we may ask the following questions about each input:

**FACTORS IN INPUT**

1) What is its form, i.e., what signals will the computer actually receive?
2) When is the input available and how does the processor know it is available? Does the processor have to request the input with a strobe signal? Does the input provide its own clock?
3) How long is it available?
4) How often does it change, and how does the processor know that it has changed?
5) Does it provide a sequence or block of data? Is the order important?
6) What should be done if the data contains errors? These may include transmission errors, incorrect data, sequencing errors, extra data, etc.
7) Is the input related to other inputs or outputs?

## DEFINING THE OUTPUTS

The next step to define is the output. We should begin by listing all the outputs that the computer must produce. Examples of outputs include:

- Data blocks to transmission lines
- Control words to peripherals
- Data to D/A converters

Then we may ask the following questions about each output:

1) What is its form, i.e., what signals must the computer produce?
2) When must it be available, and how does the peripheral know it is available?
3) How long must it be available?
4) How often must it change, and how does the peripheral know that it has changed?
5) Is there a sequence of outputs? Is the order important?
6) What should be done to avoid transmission errors or to sense and recover from peripheral failures?
7) How is the output related to other inputs and outputs?

## PROCESSING SECTION

Between the reading of input data and the sending of output results is the processing section. Here we must determine exactly how the computer must process the input data. The questions are:

**FACTORS IN PROCESSING**

1) What is the basic procedure (algorithm) for transforming input data to output results?
2) What time constraints exist? These may include data rates, delay times, the time constants of input and output devices, etc.
3) What memory constraints exist? Do we have limits on the amount of program memory or data memory, or on the size of buffers?
4) What standard programs or tables must be used? What are their requirements?
5) What special cases exist, and how should the program handle them?
6) How accurate must the results be?

## ERROR HANDLING

An important factor in many applications is the handling of errors. Clearly, the designer must make provisions for recovering from common errors and for diagnosing malfunctions. Among the questions which the designer must ask at the definition stage are:

**ERROR CONSIDERATIONS**

1) What errors could occur?
2) Which errors are most likely? If a person operates the system, human error is the most common. Following human errors, communications or transmission errors are more common than mechanical, electrical, or processor errors.
3) Which errors will not be immediately obvious to the system? A special problem is the occurrence of errors which the system or operator may not immediately recognize.
4) How can the system recover from errors with a minimum loss of time and data, and yet be aware that an error has occurred?
5) Which errors or malfunctions cause the same system behavior? How can these errors or malfunctions be separated for diagnostic purposes?
6) Which errors involve special system procedures? For example, do parity errors require re-transmission of data?

Another question is: How can the field technician systematically find the source of malfunctions without being an expert? Built-in test programs (see, for example, V.P. Srini, "Fault Diagnosis of Microprocessor Systems", Computer, January 1977, pp. 60-65), special diagnostics or signature analysis can help. For a description of signature analysis, see Gordon, G. and H. Nadig, "Hexadecimal Signatures Identify Trouble-spots in Microprocessor Systems", Electronics, March 3, 1977, pp. 89-96. There is also an Application Note (#222) entitled "A Designer's Guide to Signature Analysis" available from Hewlett-Packard.

## HUMAN FACTORS

**OPERATOR INTERACTION**

Many microprocessor-based systems involve human interaction. Human factors must be considered throughout the development process for such systems. Among the questions which the designer must ask are:

1) What input procedures will be most natural for the human operator?
2) How will an operator know where to begin, continue and end the input operation?
3) How will the operator be informed of procedural errors and equipment malfunctions?
4) What errors is the operator most likely to make?
5) How does the operator know that data has been entered correctly?
6) Are displays in a form that the operator can easily read and understand?
7) Is the response of the system adequate for the operator?
8) Is the system easy for the operator to use?
9) Are there guiding features for an inexperienced operator?
10) Are there shortcuts and reasonable options for the experienced operator?

Building a system for people to use is difficult. The microprocessor can make the system more powerful, more flexible and more responsive. However, the designer still must add the human touches which can greatly increase the usefulness and attractiveness of the system and the productivity of the human operator.

## EXAMPLES

### Response To A Switch

**DEFINING SWITCH AND LIGHT SYSTEM**

Figure 13-2 shows a simple system in which the input is from a single SPST switch and the output is to a single LED display. In response to a switch closure, the processor turns the display on for one second. Surely, this system should be easy to define.

**SWITCH AND LIGHT INPUT**

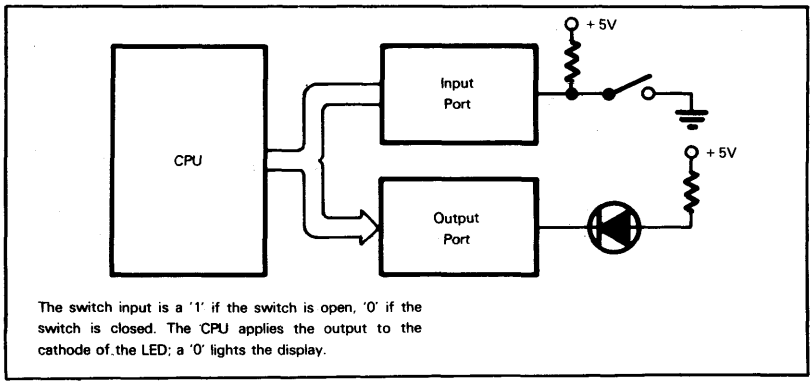
Let us first examine the input and answer each of the questions previously presented:

1) The form of the input is a single bit which may be either '0' (switch closed) or '1' (switch open).
2) The input is always available and need not be requested.
3) The input is available for at least several milliseconds after the closure.
4) The input will seldom change more than once every few seconds. The processor only has to handle the bounce in the switch. The processor must monitor the switch to determine when it is closed.
5) There is no sequence of inputs.
6) The obvious input errors are switch failure, failure in the input circuitry, and the operator attempting to close the switch again before a sufficient amount of time has elapsed. We will discuss the handling of these errors later.
7) The input does not depend on any other inputs or outputs.

The next requirement in defining the system is to examine the output. The answers to our questions are:

**SWITCH AND LIGHT OUTPUTS**

1) The form of the output is a single bit which is '0' to turn the display on, '1' to turn it off.
2) There are no time constraints on the output. The peripheral does not need to be informed of the availability of data.
3) If the display is an LED, the data need only be available for a few milliseconds at a pulse rate of about 100 times per second. The observer will see a continuously lit display.
4) The data must change after one second, i.e., go off.
5) There is no sequence of outputs.
6) The possible output errors are display failure and failure in the output circuitry.
7) The output depends only on the switch input and time.




The switch input is a '1' if the switch is open, '0' if the switch is closed. The CPU applies the output to the cathode of the LED; a '0' lights the display.

Figure 13-2. The Switch And Light System

The processing section is extremely simple. As soon as the switch input becomes a logic '0', the CPU turns the light on (a logic '0') for one second. No time or memory constraints exist.

**SWITCH AND LIGHT ERROR HANDLING**

Let us now look at the possible errors and malfunctions. These are:

1) Another switch closure before one second has elapsed.
2) Switch failure.
3) Display failure.
4) Computer failure.

Surely the first error is the most likely. The simplest solution is for the processor to ignore switch closures until one second has elapsed. This brief unresponsive period will hardly be noticeable to the human operator. Furthermore, ignoring the switch during this period means that no debouncing circuitry or software is necessary, since the system will not react to the bounce anyway.

Clearly, the last three failures can produce unpredictable results. The display may stay on, stay off or change state at random. Some possible ways to isolate the failures would be:

1) Lamp-test hardware to check the display, i.e., a button which would turn the light on independently of the processor.
2) A direct connection to the switch to check its operation.
3) A diagnostic program which would exercise the input and output circuits.

If both the display and switch are working, the computer is at fault. A field technician with proper equipment can determine the cause of the failure.

## A Switch-Based Memory Loader

**DEFINING A SWITCH-BASED MEMORY LOADER**

Figure 13-3 shows a system which allows the user to enter data into any memory location in a microcomputer. One input port, DPORT, reads data from eight toggle switches. The other input port, CPORT, is used to read control information. There are three momentary switches, HIGH ADDRESS, LOW ADDRESS and DATA. The output is the value of the last completed entry from the data switches; eight LED displays are used for the display.

The system will also, of course, require various resistors, buffers and drivers.

We shall first examine the inputs. The characteristics of the switches are the same as in the previous example; however, here there is a distinct sequence of inputs as follows:

1) The operator must set the data switches according to the eight most significant bits of an address, then
2) press the HIGH ADDRESS button. The high address bits will appear in the lights, and the program will interpret the data as the high byte of the address.
3) Then the operator must set the data switches with the value of the least significant byte of the address and
4) press the LOW ADDRESS button. The low address bits will appear in the lights, and the program will consider the data to be the low byte of the address.
5) Finally, the operator must set the desired data into the data switches and
6) press the DATA button. The display will now show the data, and the program stores the data in memory at the previously entered address.

The operator may repeat the process to enter an entire program. Clearly, even in this simplified situation, we will have many possible sequences to consider. How do we cope with erroneous sequences and make the system easy to use?

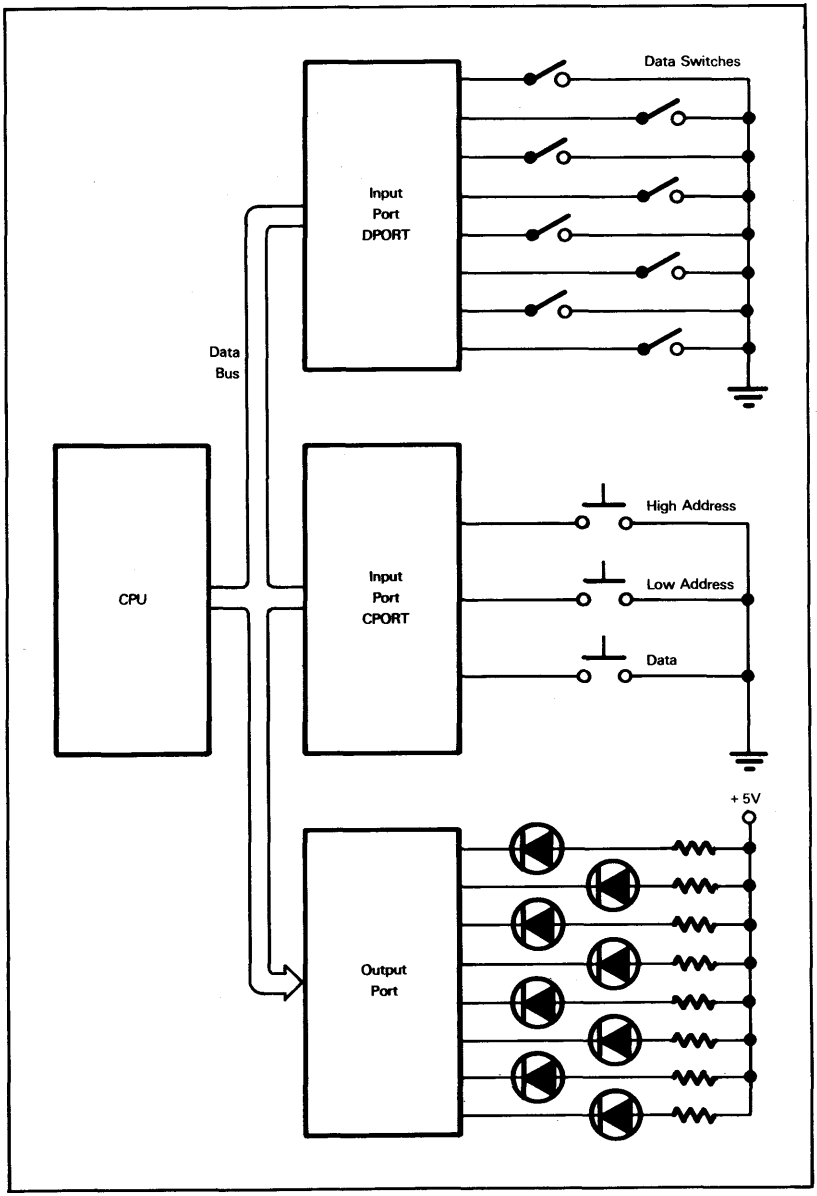



Figure 13-3. The Switch-Based Memory Loader

Output is no problem. After each input, the program sends to the displays the complement (since the displays are active-low) of the input bits. The output data remains the same until the next input operation.

The processing section remains quite simple. There are no time or memory constraints. The program can debounce the switches by waiting for a few milliseconds, and must provide complemented data to the displays.

**MEMORY LOADER ERROR HANDLING**

The most likely errors are operator mistakes. These include:

1) Incorrect entries.
2) Incorrect order.
3) Incomplete entries, e.g., forgetting the data.

The system must be able to handle these problems in a reasonable way, since they are certain to occur in actual operation.

The designer must also consider the effects of equipment failure. Just as before, the possible difficulties are:

1) Switch failure.
2) Display failure.
3) Computer failure.

In this system, however, we must pay more attention to how these failures affect the system. A computer failure will presumably cause very unusual behavior by the system, and will be easy to detect. A display failure may not be immediately noticeable; here a LAMP TEST feature will allow the operator to check the display operation. Note that we would like to test each display separately, in order to diagnose the case in which output lines are shorted together. In addition, the operator may not immediately detect switch failure; however, the operator should soon notice it and establish which switch is faulty by a process of elimination.

**OPERATOR ERROR CORRECTION IN MEMORY LOADER**

Let us look at some of the possible operator errors. Typical errors will be:

1) Erroneous data.
2) Wrong order of entries or switches.
3) Trying to go on to the next entry without completing the current one.

The operator will presumably notice erroneous data as soon as it appears on the displays. What is a viable recovery procedure for the operator? Some of the options are:

1) The operator must complete the entry procedure, i.e., enter LOW ADDRESS and DATA if the error occurs in the HIGH ADDRESS. Clearly, this procedure is wasteful and would only serve to annoy the operator.
2) The operator may restart the entry process by returning to the high address entry steps. This solution is useful if the error was in the HIGH ADDRESS, but forces the operator to re-enter earlier data if the error was in the LOW ADDRESS or DATA stage.
3) The operator may enter any part of the sequence at any time simply by setting the DATA switches with the desired data and pressing the corresponding button. This procedure allows the operator to make corrections at any point in the sequence. It also makes the DATA button a master switch which must be pressed to complete an entry. This adds a definite concluding step to each entry process.

This type of procedure should always be preferred over one that does not allow immediate error correction, has a variety of concluding steps, or enters data into the system without allowing the operator a final check. Any added complication in hardware or software will be justified in increased operator efficiency. You should always

prefer to let the microcomputer do the tedious work and recognize arbitrary sequences; it never gets tired and never forgets what was in the operating manual.

A further helpful feature would be status lights that would define the meaning of the display. Three status lights marked HIGH ADDRESS, LOW ADDRESS and DATA would let the operator know what had been entered without having to remember which button was pressed. The processor would have to monitor the sequence, but the added complication in software would simplify the operator's tasks.

We should note that, although we have emphasized human interaction, machine or system interaction has many of the same characteristics. The microprocessor should do the work. If complicating the microprocessor's task makes error recovery simple and the causes of failure obvious, the entire system will work better and be easier to maintain. Note that you should not leave consideration of system use and maintenance until the end of the software development process; instead, you should include it right in the problem-definition stage.

## A Verification Terminal

**DEFINING A VERIFICATION TERMINAL**

Figure 13-4 is a block diagram of a simple credit-verification terminal. One input port derives data from a keyboard (see Figure 13-5); the other input port accepts verification data from a transmission line. One output port sends data to a set of displays (see Figure 13-6); another sends the credit card number to the central computer. A third output port turns on one light whenever the terminal is ready to accept an inquiry, and another light when the operator sends the information. The BUSY light turns off when the response returns. Clearly, the input and output of data will be more complex than in the previous case, although the processing is still fairly simple.

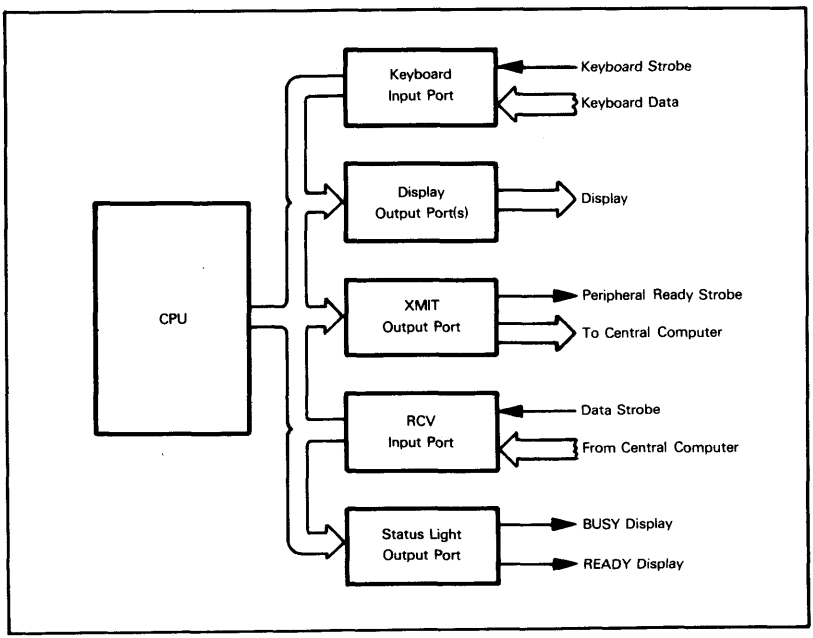
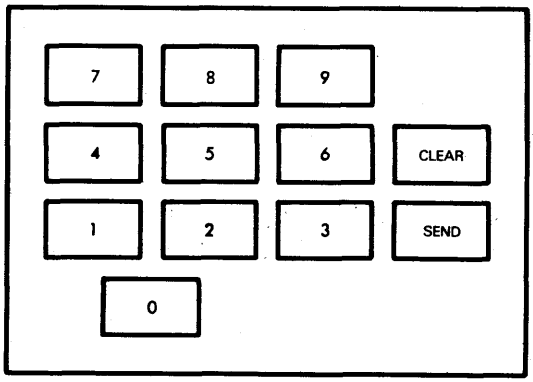
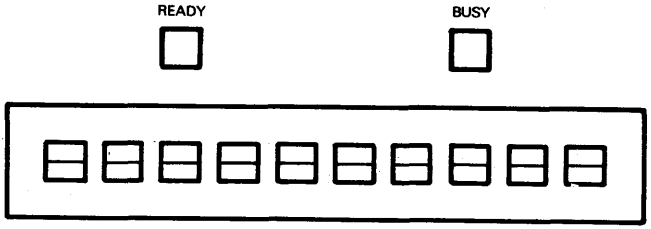



Figure 13-4. Block Diagram Of A Verification Terminal




The digit keys allow digit entries.
CLEAR deletes the entire entry.
SEND transmits the entry to the central computer.

Figure 13-5. Verification Terminal Keyboard




The display consists of ten 7-segment displays, which may be multiplexed, controlled by a shift register, or addressed separately. Two additional lights, READY and BUSY, are also present.

Figure 13-6. Verification Terminal Display

Additional displays may be useful to emphasize the meaning of the response. Many terminals use a green light for YES, a red light for NO, and a yellow light for CONSULT STORE MANAGER. Note that these lights will still have to be clearly marked with their meanings to handle the case where the operator is color-blind.

**VERIFICATION TERMINAL INPUTS**

Let us first look at the keyboard input. This is, of course, different from the switch input since the CPU must have some way of distinguishing new data. We will assume that each key closure provides a unique hexadecimal code (we can code all the keys into one digit since there are 12 keys) and a strobe. The program will have to recognize the strobe and fetch the hexadecimal number that identifies the key. There is a time constraint, since the program cannot miss any data or strobes. The constraint is not serious, since keyboard entries will be at least several milliseconds apart.

The transmission input similarly consists of a series of characters, each identified by a strobe (perhaps from a UART). The program will have to recognize each strobe and fetch the character. The data being sent across the transmission lines is usually organized into messages. A possible message format is:

1) Header.
2) Terminal destination address.
3) Coded yes or no.
4) Trailer.

The terminal will check the header, read the destination address, and see if the message is intended for it. If the message is for the terminal, the terminal accepts the data. The address could be (and often is) hard-wired into the terminal so that the terminal only receives messages intended for it. This approach simplifies the software at the cost of some flexibility.

**VERIFICATION TERMINAL OUTPUTS**

The output is also more complex than in the earlier examples. If the displays are multiplexed, the processor must not only send the data to the display port but must also direct the data to a particular display. We will need either a separate control port or a counter and decoder to handle this. Note that hardware blanking controls can blank leading zeros as long as the first digit in a multi-digit number is never zero. Software can also handle this task. Time constraints include the pulse length and frequency required to produce an apparent continuous display for the operator.

The communications output will consist of a series of characters with a particular format. The program will also have to consider the time required between characters. A possible format for the output message is:

1) Header.
2) Terminal address.
3) Credit card number.
4) Trailer.

A central communications computer may be used to poll the terminals, checking for data ready to be sent.

The processing in this system involves many new tasks, such as:

1) Identifying the control keys by number and performing the proper actions.
2) Adding the header and trailer to the outgoing message.
3) Recognizing the header and trailer in the returning message.
4) Checking the incoming terminal address.

**VERIFICATION TERMINAL ERROR HANDLING**

Note that none of the tasks involve any complex arithmetic or any serious time or memory constraints.

The number of possible errors in this system is, of course, much larger than in the earlier examples. Let us first consider the possible operator errors. These include:

1) Entering the credit card number incorrectly.
2) Trying to send an incomplete credit card number.
3) Trying to send another number while the central computer is processing one.
4) Clearing non-existent entries.

Some of these errors can be easily handled by correctly structuring the program. For example, the program should not accept the SEND key until the credit card number has been completely entered, and it should ignore any additional keyboard entries until the

response comes back from the central computer. Note that the operator will know that the entry has not been sent, since the BUSY light will not go on. The operator will also know when the keyboard has been locked out (the program is ignoring keyboard entries), since entries will not appear on the display and the READY light will be off.

**CORRECTING KEYBOARD ERRORS**

Incorrect entries are an obvious problem. If the operator recognizes an error, he or she can use the CLEAR key to make corrections. The operator will, however, make a certain number of errors without recognizing them. Most credit card numbers include a self-checking digit; the terminal could check the number before permitting it to be sent to the central computer. This step would save the central computer from wasting precious processing time checking the number.

This requires, however, that the terminal have some way of informing the operator of the error, perhaps by flashing one of the displays or by providing some other special indicator that the operator is sure to notice. Some terminals simply unlock after a maximum time delay. The operator notes that the BUSY light has gone off without an answer being received. The operator is then expected to try the entry again.

Many equipment failures are also possible. Beside the displays, keyboard and processor, there now exist the problems of communications errors or failures and central computer failures.

The data transmission will probably have to include some kind of error checking and correcting procedures. Some possibilities are:

**CORRECTING TRANSMISSION ERRORS**

1) Parity provides an error detection facility but no correction mechanism. The receiver will need some way of requesting re-transmission, and the sender will have to save a copy of the data until proper reception is acknowledged. Parity is, however, very simple to implement.
2) Short messages may use more elaborate schemes. For example, the yes/no response to the terminal could be coded so as to provide error detection and correction capability.
3) An acknowledgment and a limited number of retries could trigger an indicator which would inform the operator of a communications failure (inability to transfer a message without errors) or central computer failure (no response at all to the message within a certain period of time). Such a scheme, along with the LAMP TEST, would allow fairly simple failure diagnosis.

A communications or central computer failure indicator should also "unlock" the terminal, i.e., allow it to accept another entry. This is necessary if the terminal will not accept entries while a verification is in progress. The terminal may also unlock after a certain maximum time delay. Certain entries could be reserved for diagnostics, i.e., certain credit card numbers could be used to check the internal operation of the terminal and test the displays.

## REVIEW OF PROBLEM DEFINITION

Problem definition is as important a part of software development as it is of any other engineering task. Note that it does not require any programming or knowledge of the computer, rather, it is based on an understanding of the system and sound engineering judgment. Microprocessors can offer flexibility which the designer can use to provide a range of features which were not previously available.

Problem definition is independent of any particular computer, computer language or development system. It should, however, provide guidelines as to what type or speed of computer the application will require, and what kind of hardware/software trade-offs the designer can make. The problem definition stage is in fact really independent of

whether a computer is used at all, although a knowledge of the capabilities of the computer can help the designer in suggesting possible implementations of procedures.

# PROGRAM DESIGN

Program design is the stage in which the problem definition is formulated as a program. If the program is small and simple, this stage may involve little more than the writing of a one-page flowchart. If the program is larger or more complex, the designer may wish to consider more elaborate methods.

We will discuss flowcharting, modular programming, structured programming and top-down design. We will try to indicate the reasoning behind these methods, and their advantages and disadvantages. We will not, however, advocate any particular method, since there is no evidence that one method is always superior to all the others. You should remember that the goal is to produce a good working system, not to follow religiously the tenets of one methodology or another.

**BASIC PRINCIPLES OF PROGRAM DESIGN**

All the methodologies do, however, have some obvious principles in common. Many of these are the same principles that apply to any kind of design, such as:

1) Proceed in small steps. Do not try to do too much at one time.
2) Divide large jobs into small, logically separate tasks.
3) Keep the flow of control as simple as possible so as to make it easier to find errors.
4) Use pictorial or graphic descriptions as much as possible. They are easier to visualize than word descriptions. This is the great advantage of flowcharts.
5) Emphasize clarity and simplicity at first. You can improve performance (if necessary) once the system is working.
6) Proceed in a thorough and systematic manner. Use checklists and standard procedures.
7) Do not tempt fate. Either do not use methods that you are not sure of, or use them very carefully. Watch for situations that might cause confusion, and clarify them as soon as possible.
8) Keep in mind that the system must be debugged, tested and maintained. Plan for these later stages.
8) Keep in mind that the system must be debugged, tested and maintained. Plan for these later stages.
9) Use simple and consistent terminology and methods. Repetitiveness is no fault in program design, nor is complexity a virtue.
10) Have your design completely formulated before you start coding. Resist the temptation to start writing down instructions; it makes no more sense than making parts lists or laying out circuit boards before you know exactly what will be in the system.

## FLOWCHARTING

Flowcharting is certainly the best-known of all program design methods. Programming textbooks describe how programmers first write complete flowcharts and then start writing the actual program. In fact, few programmers have ever worked this way, and flowcharting has often been more of a joke or a nuisance to programmers than a design method. We will try to describe both the advantages and disadvantages of flowcharts, and show the place of the technique in program design.

## ADVANTAGES OF FLOWCHARTING

The basic advantage of the flowchart is that it is a pictorial representation. People find such representations much more meaningful than written descriptions. The designer can visualize the whole system and see the relationships of the various parts. Logical errors and inconsistencies often stand out instead of being hidden in a printed page. At its best, the flowchart is a picture of the entire system.

Some of the more specific advantages of flowcharts are:

1) Standard symbols exist (see Figure 13-7) so that flowcharting forms are widely recognized.
2) The flowchart may be understood by someone without a programming background.
3) The flowchart can be used to divide the entire project into sub-tasks. The flowchart can then be examined to measure overall progress.
4) The flowchart shows the sequence of operations and therefore can aid in locating the source of errors.
5) Flowcharting is a technique well-known in other areas besides programming.

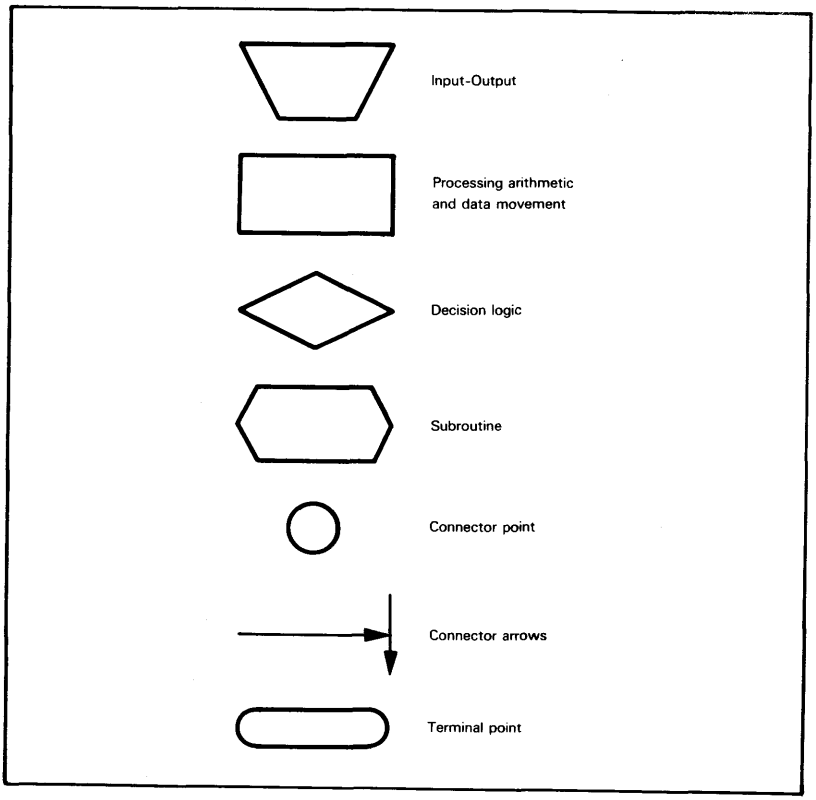



Figure 13-7. Standard Flow Diagram Symbols

These advantages are all important. There is no question that flowcharting will continue to be a widely used technique. But we should stop and look at some of the disadvantages of flowcharting as a program design method, e.g.:

**DISADVANTAGES OF FLOWCHARTING**

1) Flowcharts are difficult to design, draw or change in all except the simplest situations.
2) There is no easy way to debug or test a flowchart.
3) Flowcharts tend to become cluttered. Designers find it difficult to balance between the amount of detail needed to make the flowchart useful and the amount which makes the flowchart little better than a program listing.
4) Flowcharts only show the program organization. They do not show the organization of the data or the structure of the input/output modules.
5) Flowcharts do not help with hardware or timing problems or give hints as to where these problems might occur.

Thus, flowcharting is a helpful technique which you should not try to extend too far. Flowcharts are useful as program documentation, since they have standard forms and are comprehensible to non-programmers. As a design tool, however, flowcharts cannot provide much more than a starting outline; the programmer cannot debug a detailed flowchart and the flowchart is often more difficult to design than the program itself.

## EXAMPLES

### Response To A Switch

**FLOWCHARTING SWITCH AND LIGHT SYSTEM**

This simple task, in which a single switch turns on a light for one second, is easy to flowchart. In fact, such tasks are typical examples for flowcharting books, although they form a small part of most systems. The data structure here is so simple that it can be safely ignored.

Figure 13-8 is the flowchart. There is little difficulty in deciding on the amount of detail required. The flowchart gives a straightforward picture of the procedure which anyone could understand.

Note that the most useful flowcharts may ignore program variables and ask questions directly. Of course, compromises are often necessary here. Two versions of the flowchart are sometimes helpful — one general version in layman's language, which will be useful to non-programmers, and one programmer's version in terms of the program variables, which will be useful to other programmers.

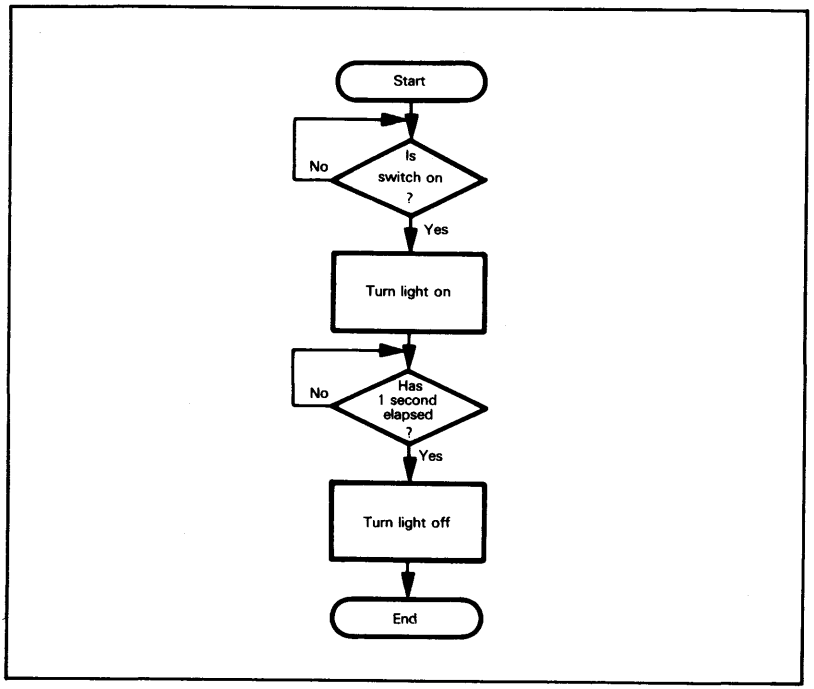



Figure 13-8. Flowchart Of One-Second Response To A Switch

## The Switch-Based Memory Loader

**FLOWCHARTING THE SWITCH-BASED MEMORY LOADER**

This system (refer to Figure 13-3) is considerably more complex than the previous example, and involves many more decisions. The flowchart (see Figure 13-9) is more difficult to write and not as straightforward as the previous example. In this example, we run into the problem that there is no way to debug or test the flowchart.

The flowchart in Figure 13-9 includes the improvements we suggested as part of the problem definition. Clearly, this flowchart is beginning to get cluttered and lose its advantages over a written description. Adding other features which define the meaning of the entry with status lights and allow the operator to check entries after completion would make the flowchart even more complex. Writing the complete flowchart from scratch could quickly become a formidable task. However, once the program has been written, the flowchart is useful as documentation.

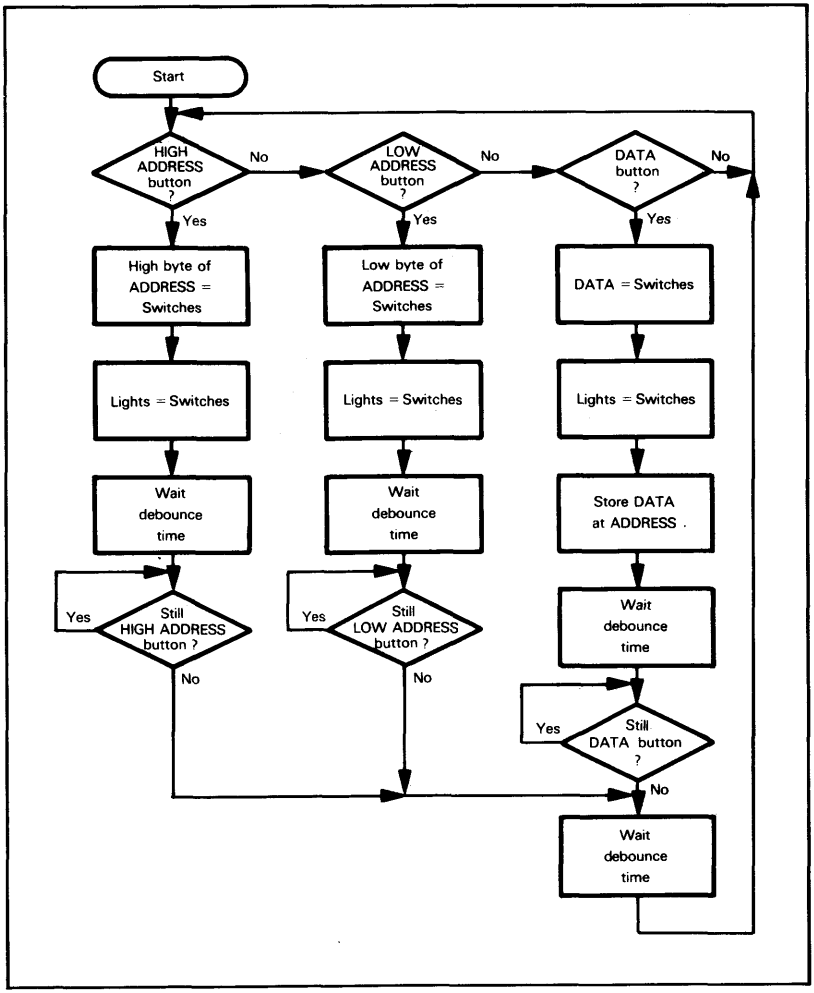



Figure 13-9. Flowchart Of Switch-Based Memory Loader

## The Credit-Verification Terminal

**FLOWCHARTING THE CREDIT VERIFICATION**

**FLOWCHARTING SECTIONS**

In this application (see Figures 13-4 through 13-6), the flowchart will be even more complex than in the switch-based memory loader case. Here, the best idea is to flowchart sections separately so that the flowcharts remain manageable. However, the presence of data structures (as in the multi-digit display and the messages) will make the gap between flowchart and program much larger.

Let us look at some of the sections. Figure 13-10 shows the keyboard entry process for the digit keys. The program must fetch the data after each strobe and place the digit in the display array if there is room for it. If there are already ten digits in the array, the program simply ignores the entry.

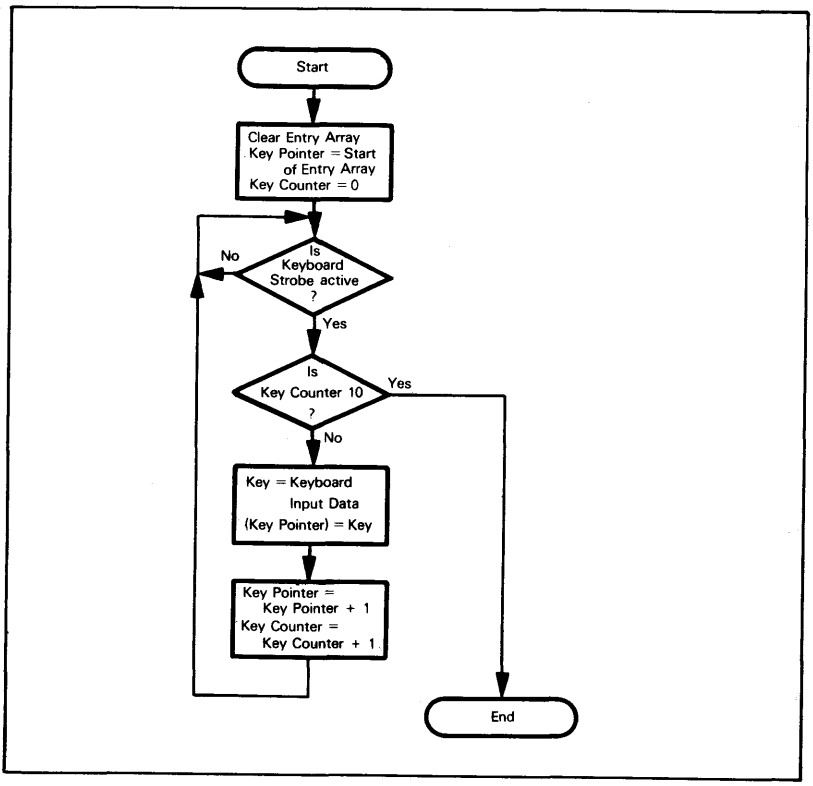



Figure 13-10. Flowchart Of Keyboard Entry Process

The actual program will have to handle the displays at the same time. Note that either software or hardware must de-activate the keyboard strobe after the processor reads a digit.

Figure 13-11 adds the SEND key. This key, of course, is optional. The terminal could just send the data as soon as the operator enters a complete number. However, that procedure would not give the operator a chance to check the entire entry. The flowchart with the SEND key is more complex because there are two alternatives:

1) If the operator has not entered ten digits, the program must ignore the SEND key and place any other key in the entry.
2) If the operator has entered ten digits, the program must respond to the SEND key by transferring control to the SEND routine, and ignore all other keys.

Note that the flowchart has become much more difficult to organize and to follow. There is also no obvious way to check the flowchart.

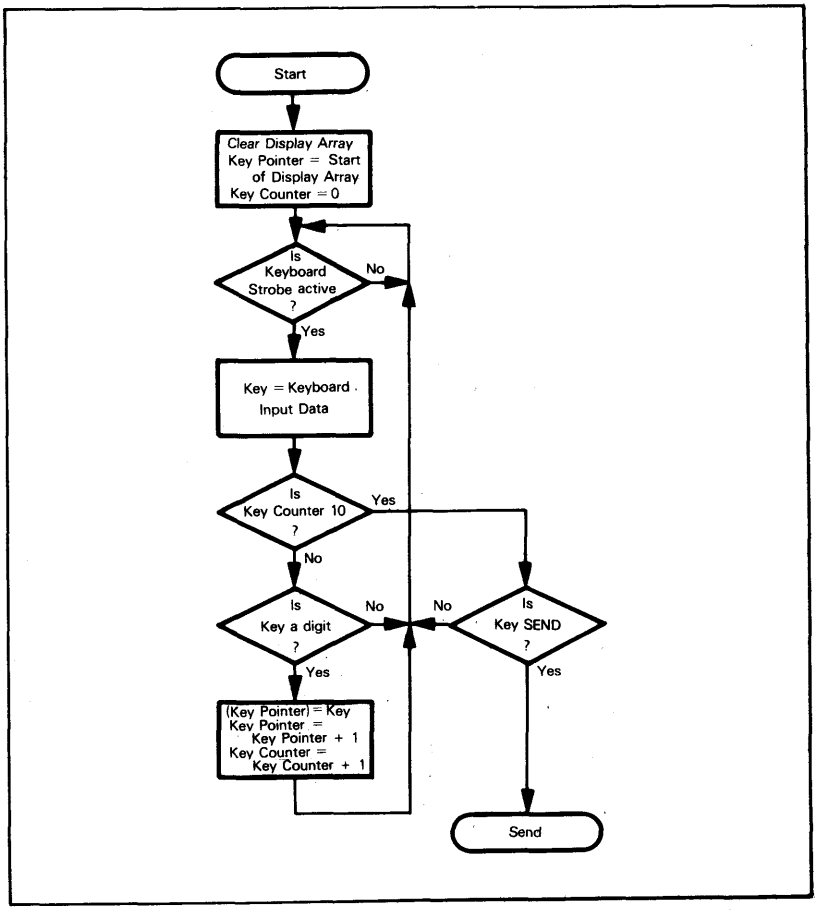



Figure 13-11. Flowchart Of Keyboard Entry Process With SEND Key

Figure 13-12 shows the flowchart of the keyboard entry process with all the function keys. In this example, the flow of control is by no means simple. Clearly, some written description is necessary. The organization and layout of complex flowcharts requires careful planning. We have followed the process of adding features to the flowchart one at a time, but this still results in a large amount of redrawing. Again we should remember that, throughout the keyboard entry process, the program must also refresh the displays if they are multiplexed and not controlled by shift registers or other hardware.

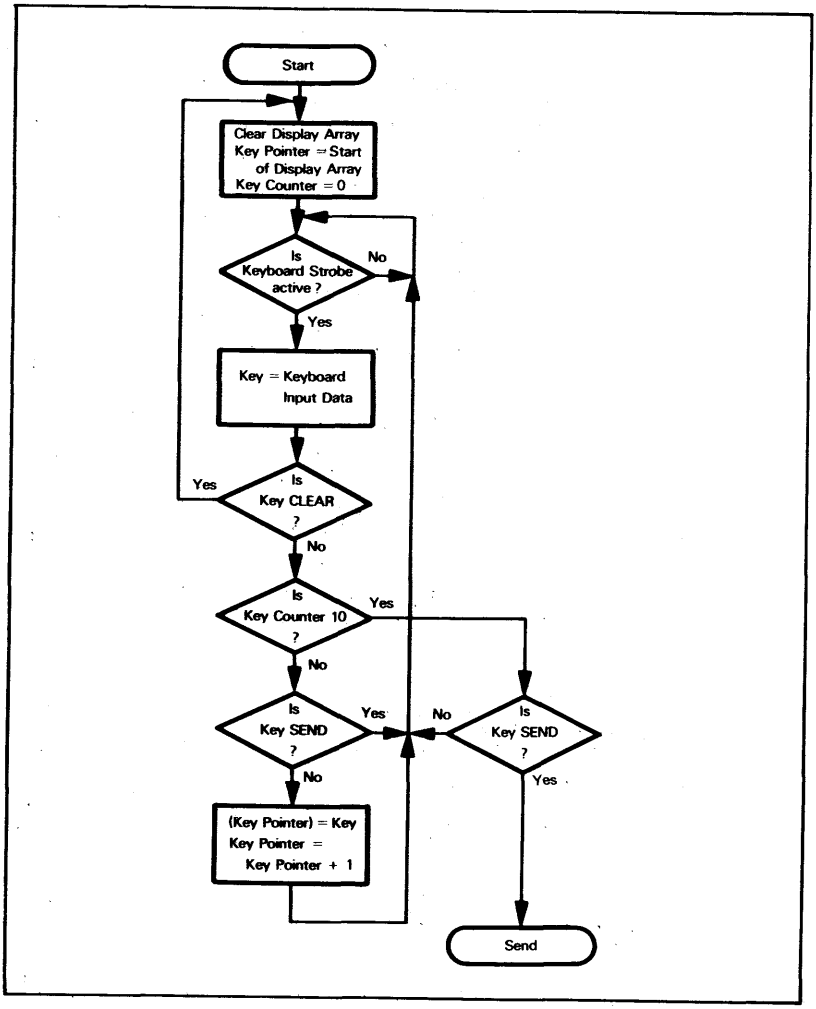



Figure 13-12. Flowchart Of Keyboard Entry Process With Function Keys

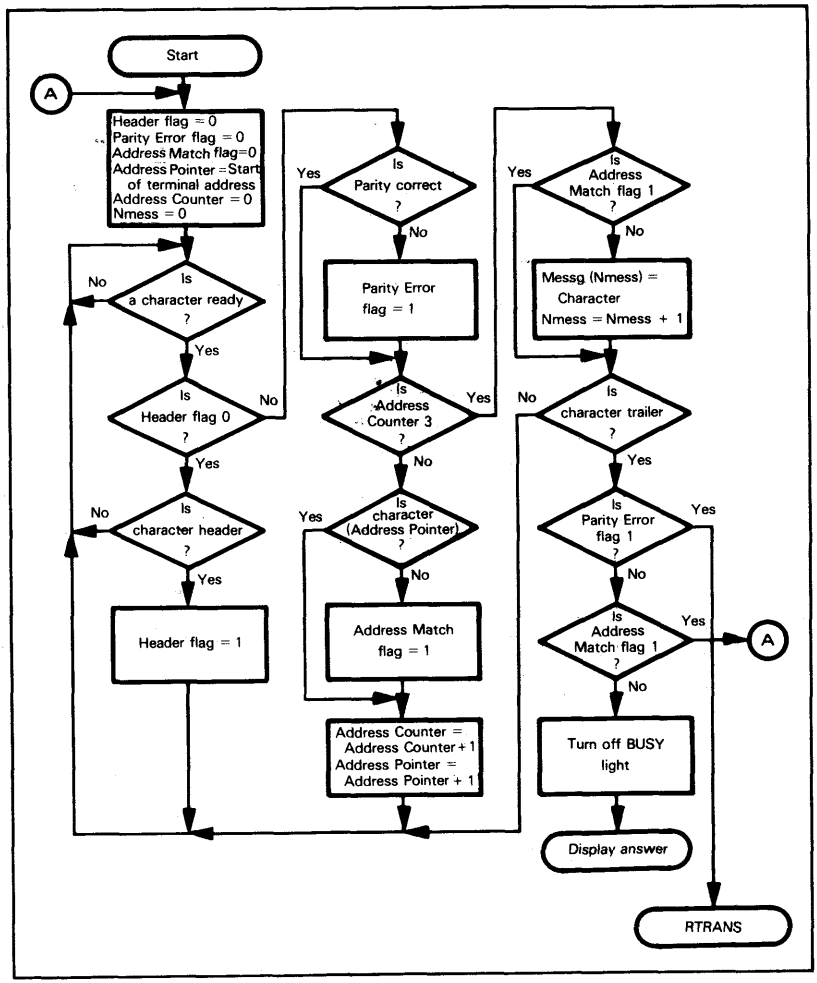



Figure 13-13. Flowchart Of Receive Routine

Figure 13-13 is the flowchart of a receive routine. We assume that the serial/parallel conversion and error checking is done in hardware (e.g., by a UART). The processor must:

1) Look for the header (we assume it is a single character).
2) Read the destination address (we assume it is three characters long) and see if the message is meant for this terminal, i.e., if the three characters agree with the terminal address.
3) Wait for the trailer character.
4) If the message is meant for the terminal, turn off the BUSY light and go to DISPLAY ANSWER routine.
5) In the event of any errors, request re-transmission by going to RTRAN routine.

This routine involves a large number of decisions, and the flowchart is neither simple nor obvious.

Clearly, we have come a long way from the simple flowchart (Figure 13-8) of the first example. A complete set of flowcharts for the transaction terminal would be a major task. It would consist of several inter-related charts with complex logic, and would require a large amount of effort. Such an effort would be just as difficult as writing a preliminary program and not as useful, since you could not check it on the computer.

## MODULAR PROGRAMS

Once programs become large and complex, flowcharting is no longer a satisfactory design tool. However, the problem definition and the flowchart can give you some idea as to how to divide the program into reasonable sub-tasks. The division of the entire program into sub-tasks or modules is called "modular programming". Clearly, most of the programs we presented in earlier chapters would typically be modules in a large system program. The problems which the designer faces in modular programming are how to divide the program into modules and how to put the modules together.

**ADVANTAGES OF MODULAR PROGRAMMING**

The advantages of modular programming are obvious:

1) A single module is easier to write, debug and test than an entire program.
2) A module is likely to be useful in many places and in other programs, particularly if it is reasonably general and performs a common task. You can build up a library of standard modules.
3) Modular programming allows the programmer to divide tasks and use previously written programs.
4) Changes can be incorporated into one module rather than into the entire system.
5) Errors can often be isolated and then attributed to a single module.
6) Modular programming gives an idea of how much progress has been made and how much of the work is left.

**DISADVANTAGES OF MODULAR PROGRAMMING**

The idea of modular programming is such an obvious one that its disadvantages are often ignored. These include:

1) Fitting the modules together can be a major problem, particularly if different people write the modules.
2) Modules require very careful documentation, since they may affect other parts of the program such as data structures used by all the modules.
3) Testing and debugging modules separately is difficult, since other modules may produce the data used by the module being debugged and still other modules may use the results. You may have to write special programs (called "drivers") just to produce sample data and test the programs. These drivers require extra programming effort which adds nothing to the total system.
4) Programs may be very difficult to modularize in any reasonable way. If you modularize the program poorly, integration will be very difficult, since almost all errors and the resulting changes will involve several modules.
5) Modular programs often require extra time and memory, since the separate modules may repeat functions.

Therefore, while modular programming is certainly an improvement over simply trying to write the entire program from scratch, it does have some disadvantages as well.

## EXAMPLES

### Response To A Switch

**MODULARIZING THE SWITCH AND LIGHT SYSTEM**

This simple program can be divided into two modules:

Module 1 waits for the switch to be turned on and turns the light on in response.

Module 2 provides the one-second delay.

Module 1 is likely to be specific to the system, since it will depend on how the switch and light are attached. Module 2 will be generally useful, since many tasks require delays. Clearly, it would be advantageous to have a standard delay module which could provide delays of varying lengths. The module will require careful documentation so that you will know how to specify the length of the delay, how to call the module, and what registers and memory locations the module affects.

A general version of Module 1 would be far less useful, since it would have to deal with different types and connections of switches and lights.

You would probably find it simpler to write a module for a particular configuration of switches and lights rather than try to use a standard routine. Note the difference between this situation and Module 2.

### The Switch-Based Memory Loader

**MODULARIZING THE SWITCH-BASED MEMORY LOADER**

The switch-based memory loader is difficult to modularize, since all the programming tasks depend on the hardware configuration and the tasks are so simple that modules hardly seem worthwhile. The flowchart in Figure 13-9 suggests that one module might be the one that waits for the operator to press one of the three pushbuttons.

Some other modules might be:

1) A delay module which provides the delay required to debounce the switches.
2) A switch and display module which reads the data from the switches and sends it to the displays.
3) A LAMP TEST module.

Such highly system-dependent modules are unlikely to be generally useful. This example is not one in which modular programming offers great advantages.

### The Verification Terminal

**MODULARIZING THE VERIFICATION TERMINAL**

The verification terminal, on the other hand, lends itself very well to modular programming. The entire system can easily be divided into three main modules:

1) Keyboard and display module.
2) Data transmission module.
3) Data reception module.

A general keyboard and display module could handle many keyboard- and display-based systems. The sub-modules would perform such tasks as:

1) Recognizing a new keyboard entry and fetching the data.
2) Clearing the array in response to a CLEAR key.
3) Entering digits into storage.
4) Looking for the terminator or SEND key.
5) Displaying the digits.

Although the key interpretations and the number of digits will vary, the basic entry, data storage and data display processes will be the same for many programs. Such function keys as CLEAR would also be standard. Clearly, the designer must consider which modules will be useful in other applications, and pay careful attention to those modules.

The data transmission module could also be divided into such sub-modules as:

1) Adding the header character.
2) Transmitting characters as the output line can handle them.
3) Generating delay times between bits or characters.
4) Adding the trailer character.
5) Checking for transmission failures, i.e., no acknowledgment or inability to transmit without errors.

The data reception module could include sub-modules which:

1) Look for the header character.
2) Check the message destination address against the terminal address.
3) Store and interpret the message.
4) Look for the trailer character.
5) Generate bit or character delays.

## REVIEW OF MODULAR PROGRAMMING

**RULES FOR MODULAR PROGRAMMING**

Modular programming can be very helpful if you abide by the following rules:

1) Use modules of 20 to 50 lines. Shorter modules are usually a waste of time, while longer modules are seldom general and may be difficult to integrate.
2) Try to make modules reasonably general. Differentiate between common features like ASCII code or asynchronous transmission formats, which will be the same for many applications and key identifications, and number of displays or number of characters in a message, which are likely to be unique to a particular application. Make the changing of the latter parameters simple. Major changes like different character codes should be handled by separate modules.
3) Take extra time on modules like delays, display handlers, keyboard handlers, etc., which will be useful in other projects or in many different places in the present program.
4) Try to keep modules as distinct and logically separate as possible.
5) Do not try to modularize simple tasks where rewriting the entire task may be easier than assembling or modifying the module.

## STRUCTURED PROGRAMMING

How do you keep modules distinct and stop them from interacting with one another? How do you write a program which has a clear sequence of operations so that you can isolate and correct errors? One answer is to use the methods known as "structured programming", whereby each part of the program consists of elements from a limited set of structures and each structure has a single entry and a single exit.

Figure 13-14 shows a flowchart of an unstructured program. If an error occurs in Module B, we have five possible sources for that error. Not only must we check each possible sequence, but we also have to make sure that any changes made to correct the error do not affect any of the other sequences. The usual result is that debugging the program becomes like trying to hang on to an octopus. Every time you think the situation is under control, there is another loose tentacle somewhere.

The answer to this problem is to establish a clear sequence of operations so that you can isolate errors. Such a clear sequence uses single-entry, single-exit modules. The basic modules that are needed are:

**BASIC STRUCTURES OF STRUCTURED PROGRAMMING**

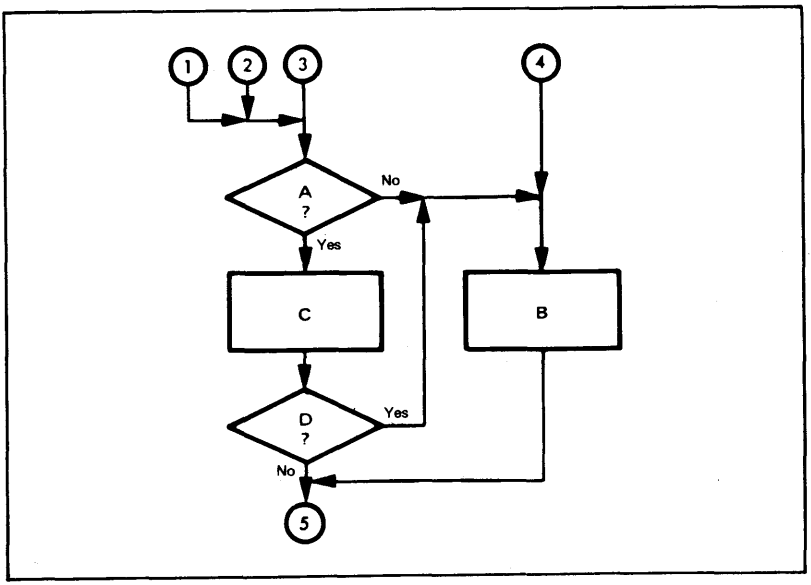



Figure 13-14. Flowchart Of An Unstructured Program

1) An ordinary sequence, i.e., a linear structure in which statements or structures are executed consecutively. In the sequence:

   P1
   P2
   P3

   the computer executes P1 first, P2 second and P3 third. P1, P2 and P3 may be single instructions or entire programs.

2) A conditional structure.

   The common one is "if A then P1 else P2", where A is a condition and P1 and P2 are programs. The computer executes P1 if A is true, and P2 if A is false. Figure 13-15 shows the logic of this structure. Note that the structure has a single entrance and a single exit; there is no way to enter or leave P1 or P2 other than through the structure.

3) A loop structure.

   The common loop structure is "do P while A", where A is a condition and P is a program. The computer checks A and executes P if A is true. This structure (see Figure 13-16) also has a single entrance and a single exit. Note that the computer will not execute P at all if A is originally false, since the value of A is checked before the execution of P.

Note the following features of structured programming:

1) Only the three basic structures are permitted.
2) Structures may be nested to any level of complexity so that any program can, in turn, contain any of the structures.
3) Each structure has a single entrance and a single exit.

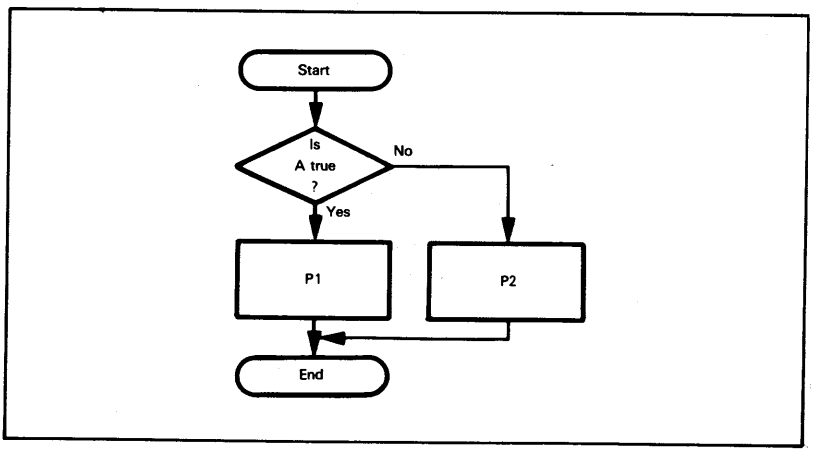



Figure 13-15. Flowchart Of The If-Then-Else Structure

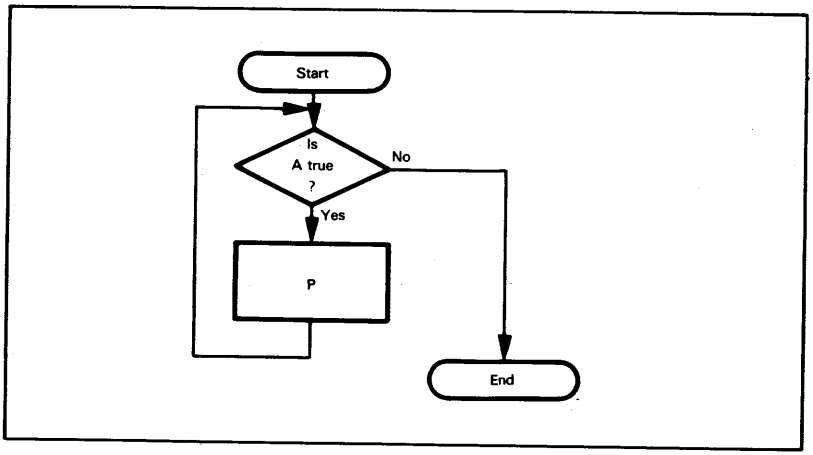



Figure 13-16. Flowchart Of The Do-While Structure

**EXAMPLES OF STRUCTURES**

Some examples of the conditional structure illustrated in Figure 13-15 are:

1) P2 included:

```
if X ≥ 0 then NPOS = NPOS+1
    else NNEG = NNEG+1
```

Both P1 and P2 are single statements.

2) P2 omitted:

```
if X ≠ 0 then Y = 1/X
```

Here no action is taken if A (X ≠ 0) is false. P2 and "else" can be omitted in this case.

Some examples of the loop structure illustrated in Figure 13-16 are:

1) Form the sum of integers from 1 to N.

```
I = 0
SUM = 0
do while I < N
    I = I+1
    SUM = SUM+I
    end
```

The computer executes the loop as long as I < N. If N = 0, the program within the "do-while" is not executed at all.

2) Count characters in an array SENTENCE until you find an ASCII period.

```
NCHAR = 0
do while SENTENCE (NCHAR) ≠ PERIOD
    NCHAR = NCHAR+1
    end
```

The computer executes the loop as long as the character in SENTENCE is not an ASCII period. The count is zero if the first character is a period.

**ADVANTAGES OF STRUCTURED PROGRAMMING**

The advantages of structured programming are:

1) The sequence of operations is simple to trace. This allows you to test and debug easily.
2) The number of structures is limited and the terminology is standardized.
3) The structures can easily be made into modules.
4) Theoreticians have proved that the given set of structures is complete, i.e., all programs can be written in terms of the three structures.
5) The structured version of a program is partly self-documenting and fairly easy to read.
6) Structured programs are easy to describe with flowcharts or other graphic methods.
7) Structured programming has been shown in practice to increase programmer productivity.

Structured programming basically forces much more discipline on the programmer than does modular programming. The result is more systematic and better-organized programs.

The disadvantages of structured programming are:

**DISADVANTAGES OF STRUCTURED PROGRAMMING**

1) Only a few high-level languages (e.g., PL/M, PASCAL) will directly accept the structures. The programmer therefore has to go through an extra translation stage to convert the structures to assembly language code. The structured version of the program, however, is often useful as documentation.
2) Structured programs often execute slower and use more memory than unstructured programs.
3) Limiting the structures to the three basic forms makes some tasks very awkward to perform. The completeness of the structures only means that all programs can be implemented with them; it does not mean that a given program can be implemented efficiently or conveniently.
4) The standard structures are often quite confusing, e.g., nested "if-then-else" structures may be very difficult to read since there may be no clear indication of where the ones inside end. A series of nested "do-while" loops can also be difficult to read.
5) Structured programs only consider the sequence of program operations, not the flow of data. Therefore, the structures may handle data awkwardly.
6) Few programmers are accustomed to structured programming. Many find the standard structures awkward and restrictive.

We are neither advocating nor discouraging the use of structured programming. It is one way of systematizing program design. In general, it is most useful in the following situations:

**WHEN TO USE STRUCTURED PROGRAMMING**

1) Larger programs, perhaps exceeding 1000 instructions.
2) Applications where memory usage is not critical.
3) Low-volume applications where software development costs, particularly testing and debugging, are important factors.
4) Applications which do not involve complex data structures.
5) Applications involving string manipulation, process control or other algorithms, rather than simple bit manipulations.
6) Applications where a high-level language is being used.

In the future we expect the cost of memory to decrease, the average size of microprocessor programs to increase, and the cost of software development to increase. Therefore, methods like structured programming, which decrease software development costs for larger programs but use more memory, will become more valuable.

## EXAMPLES

### Response To A Switch

The structured version of this example is:

**STRUCTURED PROGRAMMING IN THE SWITCH AND LIGHT SYSTEM**

```
SWITCH = OFF
do while SWITCH = OFF
  READ SWITCH
  end
LIGHT = ON
DELAY 1
LIGHT = OFF
```

ON and OFF must have the proper definitions for the switch and light. We assume that DELAY is a module that provides a wait given by its parameter in seconds.

A statement in a structured program may actually be a subroutine. However, in order to conform to the rules of structured programming, the subroutine cannot have any exits other than the one that returns control to the main program.

Since "do-while" checks the condition before executing the loop, we set the variable SWITCH to OFF before starting. The structured program is straightforward, readable, and easy to check by hand. However, it would probably require somewhat more memory than an unstructured program, which would not have to initialize SWITCH and could combine the reading and checking procedures.

## The Switch-Based Memory Loader

**STRUCTURED PROGRAMMING FOR THE SWITCH-BASED MEMORY LOADER**

The switch-based memory loader is a more complex structured programming problem. We may implement the flowchart of Figure 13-9 as follows (an * indicates a comment):

```
*
*INITIALIZE VARIABLES
*
HIADDRESS = 0
LOADDRESS = 0
DATA = 0
*
*THIS PROGRAM USES A DO-WHILE CONSTRUCT THAT HAS A CONDITION
*WHICH IS ALWAYS TRUE. THEREFORE, THE SYSTEM CONTINUALLY
*EXECUTES THE PROGRAM CONTAINED IN THIS DO-WHILE LOOP.
*
do while 1 = 1
*
*TEST FOR HIADDRESS BUTTON; PERFORM THE REQUIRED PROCESSING
*IF IT IS ON.
*
  if HIADDRBUTTON = 1 then
    HIADDRESS = SWITCHES
    LIGHTS = SWITCHES
    DELAY (DEBOUNCETIME)
    do while HIADDRBUTTON = 1
      DELAY (DEBOUNCETIME)
      end
*
*TEST FOR LOADDRESS BUTTON; PERFORM LO ADDRESS PROCESSING IF
*IT IS ON.
*
  if LOADDRBUTTON = 1 then
    LOADDRESS = SWITCHES
    LIGHTS = SWITCHES
    DELAY (DEBOUNCETIME)
    do while LOADDRBUTTON = 1
      DELAY (DEBOUNCETIME)
      end
```

```
*
*TEST FOR DATABUTTON, AND STORE DATA INTO MEMORY IF IT
*IS ON
*
  if DATABUTTON = 1 then
    DATA = SWITCHES
    LIGHTS = SWITCHES
    (HIADDRESSLOADDRESS) = DATA
    DELAY (DEBOUNCETIME)
    do while DATABUTTON = 1
      DELAY (DEBOUNCETIME)
      end
  end
*
*THE ABOVE END TERMINATES THE
*  do while 1 = 1 LOOP.
*
```

Structured programs are not easy to write, but they can give a great deal of insight into the overall program logic. You can check the logic of the structured program by hand before writing any actual code.

## The Credit-Verification Terminal

**STRUCTURED PROGRAM FOR THE CREDIT-VERIFICATION TERMINAL**

**STRUCTURED KEYBOARD ROUTINE**

Let us look at the keyboard entry for the transaction terminal. We will assume that the display array is ENTRY, the keyboard strobe is KEYSTROBE, and the keyboard data is KEYIN. The structured program without the function keys is:

```
NKEYS = 10
*
*CLEAR ENTRY TO START
*
  do while NKEYS > 0
    NKEYS = NKEYS - 1
    ENTRY(NKEYS) = 0
    end
*
*FETCH A COMPLETE ENTRY FROM KEYBOARD
*
  do while NKEYS < 10
    if KEYSTROBE = ACTIVE then
      KEYSTROBE = INACTIVE
      ENTRY(NKEYS) = KEYIN
      NKEYS = NKEYS+1
    end
```

Adding the SEND key means that the program must ignore extra digits after it has a complete entry, and must ignore the SEND key until it has a complete entry. The structured program is:

```
NKEYS = 10
*
*CLEAR ENTRY TO START
*
  do while NKEYS > 0
    NKEYS = NKEYS - 1
    ENTRY(NKEYS) = 0
    end
```

```
*
*WAIT FOR COMPLETE ENTRY AND SEND KEY
*
  do while KEY ≠ SEND OR NKEYS ≠ 10
    if KEYSTROBE = ACTIVE then
      KEYSTROBE = INACTIVE
      KEY = KEYIN
*
*GET DIGIT IF COMPLETE ENTRY NOT PRESENT
*
    if NKEYS ≠ 10 AND KEY ≠ SEND then
      ENTRY(NKEYS) = KEY
      NKEYS = NKEYS+1
    end
```

Adding the CLEAR key allows the program to clear the entry originally by simulating a voiding, i.e., by setting NKEYS to 10 and KEY to CLEAR before starting. The structured program must also only clear digits which have previously been filled. The new structured program is:

```
*
*SIMULATE COMPLETE CLEARING
*
NKEYS = 10
KEY = CLEAR
*
*WAIT FOR COMPLETE ENTRY AND SEND KEY
*
do while  KEY ≠ SEND OR NKEYS ≠ 10
*
*CLEAR WHOLE ENTRY IF CLEAR KEY STRUCK
*
  if KEY = CLEAR then
    KEY = 0
    do while NKEYS > 0
      NKEYS = NKEYS - 1
      ENTRY(NKEYS) = 0
      end
*
*GET DIGIT IF ENTRY INCOMPLETE
*
  if KEYSTROBE = ACTIVE then
    KEYSTROBE = INACTIVE
    KEY = KEYIN
    if KEY < 10 AND NKEYS ≠ 10 then
      ENTRY(NKEYS) = KEY
      NKEYS = NKEYS+1
  end
```

Note that the program clears the CLEAR key so that it only carries out the voiding process once.

**STRUCTURED RECEIVE ROUTINE**

We can similarly build a structured program for the receive routine. An initial program could just look for the header and trailer characters. We will assume that RSTB is the indicator that a character is ready. The structured program is:

```
*
*CLEAR HEADER FLAG TO START
*
HFLAG = 0
*
*WAIT FOR HEADER AND TRAILER
*
do while HFLAG = 0 OR CHAR ≠ TRAILER
*
*GET CHARACTER IF READY, LOOK FOR HEADER
*
  if RSTB = ACTIVE then
    RSTB = INACTIVE
    CHAR = INPUT
    if CHAR = HEADER then
      HFLAG = 1
  end
```

Now we can add the section that checks the message address against the three digits in TERMINAL ADDRESS (TERMADDR). If any of the corresponding digits are not equal, the ADDRESS MATCH flag (ADDRMATCH) is set to 1.

```
*
*CLEAR HEADER FLAG, ADDRESS MATCH FLAG, ADDRESS COUNTER TO
*START
*
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
*
*WAIT FOR HEADER, DESTINATION ADDRESS AND TRAILER
*
do while HFLAG = 0 OR CHAR ≠ TRAILER OR ADDRCTR ≠ 3
*
*GET CHARACTER IF READY
*
  if RSTB = ACTIVE then
    RSTB = INACTIVE
    CHAR = INPUT
*
*CHECK FOR TERMINAL ADDRESS AND HEADER
*
  if HFLAG = 1 AND ADDRCTR ≠ 3 then
    if CHAR ≠ TERMADDR(ADDRCTR) then ADDRMATCH = 1
    ADDRCTR = ADDRCTR+1
  if CHAR = HEADER then HFLAG = 1
  end
```

The program must now wait for a header, a three-digit identification code, and a trailer. You must be careful of what happens during the iteration when the program finds the header, and of what happens if an erroneous identification code character is the same as the trailer.

A further addition can store the message in MESSG. NMESS is the number of characters in the message; if it is not zero at the end, the program knows that the terminal has received a valid message. We have not tried to minimize the logic expressions in this program.

```
*
*CLEAR FLAGS, COUNTERS TO START
*
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
NMESS = 0
*
*WAIT FOR HEADER, DESTINATION ADDRESS AND TRAILER
*
do while HFLAG = 0 OR CHAR ≠ TRAILER OR ADDRCTR ≠ 3
*
*GET CHARACTER IF READY
*
  if RSTB = ACTIVE then
    RSTB = INACTIVE
    CHAR = INPUT
*
*READ MESSAGE IF DESTINATION ADDRESS = TERMINAL ADDRESS
*
  if HFLAG = 1 AND ADDRCTR = 3 then
    if ADDRMATCH = 0 AND CHAR ≠ TRAILER then
      MESSG(NMESS) = CHAR
      NMESS = NMESS+1
*
*CHECK FOR TERMINAL ADDRESS
*
  if HFLAG = 1 AND ADDRCTR ≠ 3 then
    if CHAR ≠ TERMADDR(ADDRCTR) then ADDRMATCH = 1
    ADDRCTR = ADDRCTR+1
*
*LOOK FOR HEADER
*
  if CHAR = HEADER then HFLAG = 1
end
```

The program only checks for the identification code if it found a header during a previous iteration. It only accepts the message if it has previously found a header and a complete, matching destination address. The program must work properly during the iterations when it finds the header, the trailer and the last digit of the destination address. It must not try to match the header with the terminal address or place the trailer or the final digit of the destination address in the message. You might try adding the rest of the logic from the flowchart (Figure 13-13) to the structured program. Note that the order of operations is often critical. You must be sure that the program does not complete one phase and start the next one during the same iteration.

## REVIEW OF STRUCTURED PROGRAMMING

Structured programming brings discipline to program design. It forces you to limit the types of structures you use and the sequence of operations. It provides single-entrance, single-exit structures which you can check for logical accuracy. Structured programming often makes the designer aware of inconsistencies or possible combinations of inputs. Structured programming is not a cure-all, but it does bring some order into a pro-

cess that can be chaotic. The structured program should also aid in debugging, testing and documentation.

Structured programming is not simple. The programmer must not only define the problem adequately, but must also work through the logic carefully. This is tedious and difficult, but it can result in a clearly written, working program.

**TERMINATORS FOR STRUCTURES**

The particular structures we have presented are not ideal and are often awkward. In addition, it can be difficult to distinguish where one structure ends and another begins, particularly if they are nested. Theorists may provide better structures in the future, or designers may wish to add some of their own. Some kind of terminator for each structure seems necessary, since indenting does not always clarify the situation. "End" is a logical terminator for the "do-while" loop. There is no obvious terminator, however, for the "if-then-else" statement; some theorists have suggested "endif" or "fi" ("if" backwards), but these are both awkward and do not contribute to the readability of the program.

**RULES FOR STRUCTURED PROGRAMMING**

We suggest the following rules for applying structured programming:

1) Begin by writing a basic flowchart to help define the logic of the program.
2) Start with the "if-then-else" and "do-while" constructs. They are known to be a complete set, i.e., any program can be written in terms of these structures.
3) Indent each level a few spaces so that you will know which statements belong where.
4) Use terminators for each structure, e.g., "end" for the "do-while" and "endif" or "fi" for the "if-then-else". The terminators plus the indentation should make the program reasonably clear.
5) Emphasize simplicity and readability. Leave lots of spaces, use meaningful names, and make expressions as clear as possible. Do not try to minimize the logic at the cost of clarity.
6) Comment the program in an organized manner.
7) Check the logic. Try all the extreme cases or special conditions and a few sample cases. Any logical errors you find at this level will not return to plague you later.

## TOP-DOWN DESIGN

**BOTTOM-UP DESIGN**

The remaining problem is how to check and integrate modules or structures. Certainly we want to divide a large task into sub-tasks. But how do we check the sub-tasks in isolation and put them together? The standard procedure, called "bottom-up design", requires extra work in testing and debugging and leaves the entire integration task to the end. What we need is a method that will allow testing and debugging to occur in the actual program environment and will subdivide the system integration into a series of modular tasks.

**TOP-DOWN DESIGN METHODS**

**PROGRAM STUBS**

This method is "top-down design". Here we start by writing the overall supervisor program. We replace the undefined sub-programs by program "stubs", temporary programs which may either record the entry, provide the answer to a selected test problem, or do nothing. We then test the supervisor program to see that its logic is correct.

We proceed by expanding the stubs. Each stub will often contain sub-tasks, which we will temporarily represent as stubs. This process of expansion, debugging and testing continues until all the stubs are replaced by working programs. Note that testing and integration occur at each level, rather than all at the end. No special driver or data generation programs are necessary. We get a clear idea of exactly where we are in the design. Top-down design assumes modular programming, and is compatible with structured programming as well.

**EXPANDING PROGRAM STUBS**

**ADVANTAGES OF TOP-DOWN DESIGN**

The disadvantages of top-down design are:

**DISADVANTAGES OF TOP-DOWN DESIGN**

1) The overall design may not mesh well with system hardware.
2) It may not take good advantage of existing software.
3) A suitable stub may be difficult to write, particularly if the same stub must work correctly in several different places.
4) Top-down design may not result in generally useful modules.
5) Errors at the top level can have catastrophic effects, whereas errors in bottom-up design are usually limited to a particular module.

In large programming projects, top-down design has been shown to greatly improve programmer productivity. However, almost all of these projects have used some bottom-up design in cases where the top-down method would have resulted in a large amount of extra work.

Top-down design is a useful tool which need not be followed to extremes. It provides the same discipline for system testing and integration that structured programming provides for module design. The method, however, has more general applicability since it does not really assume the use of programmed logic. However, top-down design may not result in the most efficient implementation.

## EXAMPLES

### Response To A Switch

**TOP-DOWN DESIGN OF SWITCH AND LIGHT SYSTEM**

The first structured programming example actually demonstrates top-down design as well. The program was:

```
SWITCH = OFF
do while SWITCH = OFF
  READ SWITCH
  end
LIGHT = ON
DELAY 1
LIGHT = OFF
```

Almost all of these statements are really stubs, since none of them is fully defined. For example, what does READ SWITCH mean? If the switch were one bit of input port SPORT, it really means

```
SWITCH = SPORT AND SMASK
```

where SMASK has a '1' bit in the appropriate position.

Similarly, DELAY 1 actually means (if the processor itself provides the delay):

```
REG = COUNT
do while REG ≠ 0
  REG = REG - 1
  end
```

COUNT is the appropriate number to provide a one-second delay. The expanded version of the program is:

```
SWITCH = 0
do while SWITCH = 0
  SWITCH = SPORT AND MASK
  end
LIGHT = ON
REG = COUNT
do while REG ≠ 0
  REG = REG - 1
  end
LIGHT = NOT (LIGHT)
```

Certainly this program is more explicit, and could more easily be translated into actual instructions or statements.

## The Switch-Based Memory Loader

**TOP-DOWN DESIGN OF SWITCH-BASED MEMORY LOADER**

This example is more complex than the first example, so that we must proceed more systematically. Here again, the structured program really contains stubs.

For example, if the HIGH ADDRESS button is one bit of input port CPORT, "if HIADDRBUTTON = 1" really means:

1) Input from CPORT.
2) Complement.
3) Logical AND with HAMASK;

where HAMASK has a '1' in the appropriate bit position and '0s' elsewhere. Similarly, the condition "if DATABUTTON = 1" really means:

1) Input from CPORT.
2) Complement.
3) Logical AND with DAMASK.

So, the initial stubs could just assign values to the buttons, e.g.,

```
HIADDRBUTTON = 0
LOADDRBUTTON = 0
DATABUTTON = 0
```

A run of the supervisor program should show that it takes the implied "else" path through the "if-then-else" structures, and never reads the switches. Similarly, if the stub were

```
HIADDRBUTTON = 1
```

the supervisor program should stay in the "do while HIADDRBUTTON = 1" loop waiting for the button to be released. These simple runs check the overall logic.

Now we can start expanding each stub and seeing if the expansion produces a reasonable overall result. Note how debugging and testing proceed in a straightforward and modular manner. We expand the HIADDRBUTTON = 1 stub to

```
READ CPORT
HIADDRBUTTON = NOT (CPORT) AND HAMASK
```

The program should wait for the HIGH ADDRESS button to be closed. The program should then place the contents of the switches in the lights. This run checks for the proper response to the HIGH ADDRESS button.

We then expand the LOW ADDRESS button module to

```
READ CPORT
HIADDRBUTTON = NOT (CPORT) AND LAMASK
```

With the LOW ADDRESS button in the closed position, the program should place the contents of the switches in the lights. This run checks for the proper response to the LOW ADDRESS button.

Similarly, we can expand the DATA button module and check for the proper response to that button. The entire program will then have been tested in a systematic manner.

When all the stubs have been expanded, the coding, debugging and testing stages will all be complete. Of course, we must know exactly what results each stub should produce. However, many logical errors will become obvious at each level without any further expansion.

## The Transaction Terminal

**TOP-DOWN DESIGN OF VERIFICATION TERMINAL**

This example, of course, will have more levels of detail. We could start with the following program (see Figure 13-17 for a flowchart):

```
KEYBOARD
ACK = 0
do while ACK = 0
  TRANSMIT
  RECEIVE
  end
DISPLAY
```

Here KEYBOARD, TRANSMIT, RECEIVE and DISPLAY are program stubs which will be expanded later. KEYBOARD, for example, could simply place a ten-digit verified number in the appropriate buffer.

**EXPANDING THE KEYBOARD ROUTINE**

The next stage of expansion could produce the following program for KEYBOARD (see Figure 13-18):

```
VER = 0
do while VER = 0
  COMPLETE = 0
do while COMPLETE = 0
  KEYIN
  KEYDS
  end
VERIFY
end
```

Here VER = 0 means that an entry has not been verified; COMPLETE = 0 means that the entry is incomplete. KEYIN and KEYDS are the keyboard input and display routines respectively. VERIFY checks the entry. A possible stub for KEYIN would simply place a random entry (from a random number table or generator) in the buffer and set COMPLETE to 1.

We would continue by similarly expanding, debugging and testing TRANSMIT, RECEIVE and DISPLAY. Note that you should expand each program by one level so that you do not perform the integration of an entire program at any one time. You must use your judgment in defining levels. Too small a step wastes time, while too large a step gets you back to the problems of system integration which you want to avoid.

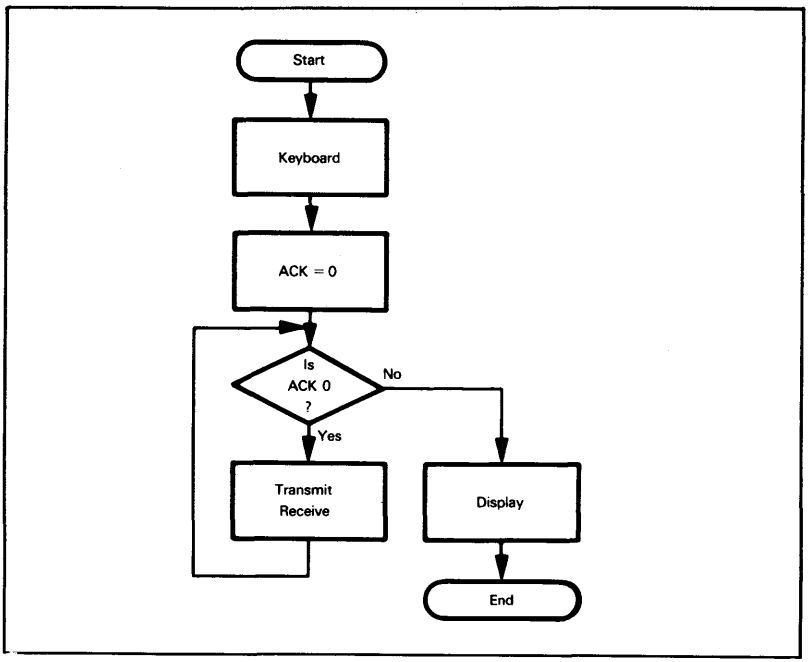



Figure 13-17. Initial Flowchart For Transaction Terminal

## REVIEW OF TOP-DOWN DESIGN

Top-down design brings discipline to the testing and integration phases of program design. It provides a systematic method for expanding a flowchart or problem definition to the level required to actually write a program. Together with structured programming, it forms a complete set of design techniques.

Like structured programming, top-down design is not simple. The designer must have defined the problem carefully and must work systematically through each level. Here again the methodology may seem tedious, but the payoff can be substantial if the designer really follows the rules.

**FORMAT FOR TOP-DOWN DESIGN**

We recommend the following approach to top-down design:

1) Start with a basic flowchart.
2) Make the stubs as complete and as separate as possible.
3) Define precisely all the possible outcomes from each stub and select a test set.
4) Check each level carefully and systematically.
5) Use the structures from structured programming.
6) Expand each stub by one level. Do not try to do too much in one step.
7) Watch carefully for common tasks and data structures.
8) Test and debug after each stub expansion. Do not try to do an entire level at a time.
9) Be aware of what the hardware can do. Do not hesitate to stop and do a little bottom-up design where that seems necessary.

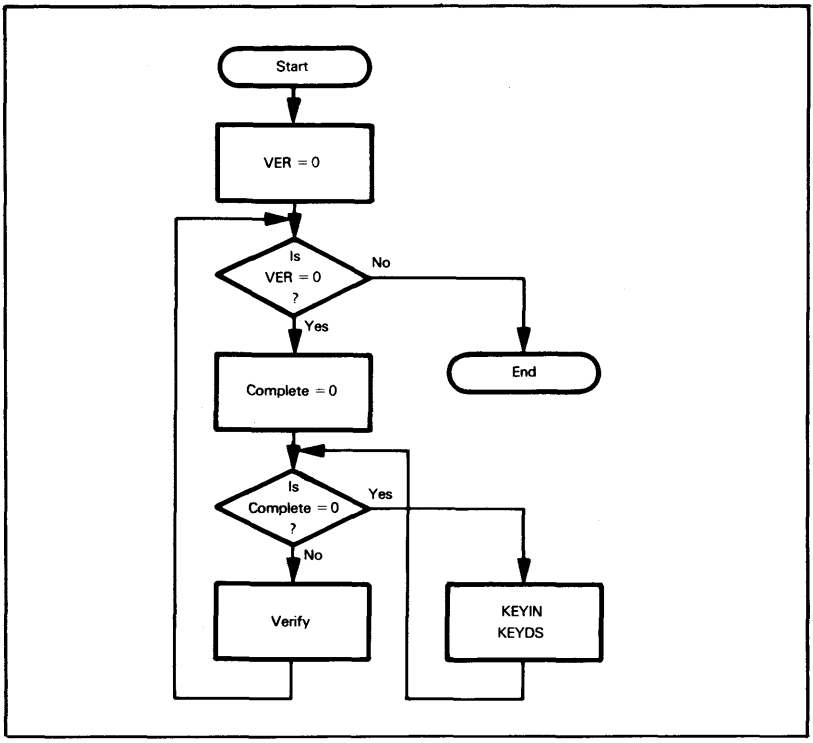



Figure 13-18. Flowchart For Expanded KEYBOARD Routine

## REVIEW OF PROBLEM DEFINITION AND PROGRAM DESIGN

You should note that we have spent an entire chapter without mentioning any specific microprocessor or assembly language, and without writing a single line of actual code. Hopefully, though, you now know a lot more about the examples than you would have if we had just asked you to write the programs at the start. Although we often think of the writing of computer instructions as a key part of software development, it is actually one of the simplest stages.

Once you have written a few programs, most of the methodology will become simple. You will soon learn the instruction set, recognize which instructions are really useful, and remember the common sequences that make up the major part of most programs. You will then find that many of the other stages of software development remain difficult and have few clear rules.

We have suggested here some ways to systematize the important early stages. In the problem definition stage, you must define all the characteristics of the system — its inputs, outputs, processing, time and memory constraints, and error handling. You must particularly consider how the system will interact with the larger system of which it is a part, and whether that larger system includes electrical equipment, mechanical equipment or human operator. You must start at this stage to consider how to make the system easy to use and maintain.

In the program-design stage, several techniques can help you to systematically specify and document the logic of your program. Modular programming forces you to divide the total program into reasonably sized modules. Structured programming provides a systematic way of defining the logic of those modules, while top-down design is a systematic method for integrating and testing them. Of course, no one can compel you to follow all of these techniques; they are, in fact, guidelines more than anything else. But they do provide a unified approach to the definition and design stages, and you should consider them a basis on which to develop your own approach.

## REFERENCES


1. Hughes, J.L. and J.I. Michtom, A Structured Approach to Programming, Prentice-Hall, Englewood Cliffs, N.J., 1977.
2. Leventhal, L.A., "Can Structured Programming Help the Bench Programmer?", IEEE Workshop on Bench Programming of Microprocessors, Philadelphia, PA., June 1977.
3. Ulrickson, R.W., "Software Modules are the Building Blocks", Electronic Design, February 1, 1977, pp. 62-66.
4. Ulrickson, R.W., "Solve Software Problems Step-by-Step", Electronic Design, January 18, 1977, pp. 54-58.
5. Yourdon, E.U., Techniques of Program Structure and Design, Prentice-Hall, Englewood Cliffs, N.J., 1975.

# Chapter 14
# DEBUGGING AND TESTING

As we noted at the beginning of the previous chapter, debugging and testing are among the most time-consuming stages in software development. Even though such methods as modular programming, structured programming and top-down design can reduce the frequency of errors and simplify testing, debugging and testing still are difficult because they are such poorly-defined tasks. Many textbooks have been written about designing and coding programs. Very little has been written about how to debug and test programs. The selection of an adequate set of test data is seldom a clear or scientific process: Finding errors sometimes seems like a game of "pin the tail on the donkey", except that the donkey is moving and the programmer must position the tail by remote control. Surely, few tasks are as frustrating as debugging programs.

This chapter will first describe the tools available for debugging. It will then discuss the basic debugging procedure, describe the common types of errors and show some examples of program debugging. The last sections will describe how to select test data and test programs.

We will not do much more than describe the purpose of most debugging tools. There is very little standardization in this area and not enough space to discuss all of the devices and programs available on the market. The examples should give you some idea as to when and why particular hardware or software will be helpful.

## SIMPLE DEBUGGING TOOLS

The simplest debugging tools available are:

1) A single-step facility
2) A breakpoint facility
3) A register dump program (or utility)
4) A memory dump program

**SINGLE-STEP**

The single-step facility simply allows you to execute the program one step at a time. Most Intel 8080-based microcomputers have this ability since the circuitry is quite simple. Of course, the only things you will be able to see when you execute a single-step are the states of the output lines which you are monitoring. The most important lines are:

1) Data Bus
2) Address Bus
3) Status latch outputs
4) Control lines

If you monitor these lines (either in hardware or software), you will be able to see the addresses, instructions and data appear as the program executes. You will be able to tell what kinds of operations the CPU is performing by examining these lines at appropriate times. This information will be sufficient to inform you of such errors as incorrect Jump instructions, omitted or incorrect addresses, erroneous operation codes or incorrect data values. However, you cannot see the contents of registers and flags without some additional debugging facility or a special sequence of instructions. Therefore, much of the logic of the program may remain invisible.

**LIMITATIONS OF SINGLE-STEP MODE**

There are many errors which a single-step mode cannot find. These include timing errors and errors in the interrupt or DMA systems. Furthermore, the single-step mode is very slow, typically running at less than one millionth of the speed of the processor itself. Therefore, to single-step through one second of real processor time would take more than ten days. Clearly, single-stepping is only useful to check the logic of short instruction sequences.

**BREAKPOINT**

A breakpoint is an address at which the program will automatically halt or otherwise stop its normal routine so that you can examine the current status of the system. The program will usually not start again until you clear the breakpoint. The breakpoint allows you to check or pass through an entire section of a program. Thus, to see if an entire initialization routine is correct, you can place a breakpoint at the end of the initialization and run the program. You can then check the status of memory locations and registers to see if the entire section is correct.

Breakpoints complement the single-step mode. You can use the breakpoint either to localize the error or to pass through sections which you know are correct. You can then do the detailed debugging in the single-step mode. Breakpoints do not usually affect the timing of the program, so they can be used to check input/output and interrupts.

**RST AS A BREAKPOINT**

Breakpoints often take advantage of part or all of the microprocessor interrupt system. Some micros have a special SOFTWARE INTERRUPT or TRAP facility which can act as a breakpoint. On the Intel 8080, if you are not already using all the interrupt vectors in your program, you can use the RST (Restart) instruction as a breakpoint. Table 14-1 gives the destination addresses for the various RST instructions. Chapter 12 describes the RST instruction in more detail. The breakpoint routine can print register and memory contents or just wait in place (HLT or a conditional or unconditional jump to itself) until the user allows the machine to proceed. But remember that the RST instruction uses the Stack and Stack Pointer to store the return address. Figure 14-1 shows a routine where RST 3 results in an endless loop. The programmer would have to clear this breakpoint with a RESET or interrupt signal.

```
        ORG     18H
RST3    EQU     18H
        JMP     RST3        ;WAIT IN PLACE
```

Figure 14-1. A Simple Breakpoint Routine

A more powerful facility could allow you to enter an address to which the processor would transfer control. Another possibility would be a return dependent on a switch:

```
        ORG     18H
RST3    EQU     18H
        PUSH    PSW         ;SAVE A
WAITS:  IN      SPORT       ;WAIT FOR SWITCH = 1
        ANI     MASK
        JZ      WAITS
        POP     PSW         ;RESTORE A
        RET
```

Monitor programs and development systems will generally provide some kind of useful breakpoint facility.

Table 14-1. Intel 8080 Restart Addresses

| Instruction (Mnemonic) | (Hex) | Destination Address (Hex) |
|---|---|---|
| RST 0 | C7 | 0000 |
| RST 1 | CF | 0008 |
| RST 2 | D7 | 0010 |
| RST 3 | DF | 0018 |
| RST 4 | E7 | 0020 |
| RST 5 | EF | 0028 |
| RST 6 | F7 | 0030 |
| RST 7 | FF | 0038 |

**REGISTER DUMP**

A register dump utility on a microcomputer is a program which will display the contents of all the CPU registers. The following routine will display the contents of the registers if we assume PRNT displays the contents of the Accumulator as two hexadecimal digits.

Figure 14-2 is a flowchart of the program and Figure 14-3 shows a typical result.

```
;
;PLACE ALL REGISTER CONTENTS IN STACK
;
        PUSH    PSW     ;PC ALREADY ON STACK
        PUSH    B
        PUSH    D       ;SAVE REGISTERS IN STACK
        PUSH    H
;
;USE STACK POINTER AS STARTING ADDRESS
;
        LXI     H,0
        DAD     SP      ;GET STACK POINTER
        LXI     D,10    ;COMPUTE VALUE OF ORIGINAL SP
        DAD     D
        PUSH    H       ;SAVE ORIGINAL STACK POINTER
;
;PRINT CONTENTS OF REGISTERS
;ORDER IS PC (HIGH), PC (LOW), A, FLAGS, B, C, D, E, H, L, SP
;(HIGH), SP (LOW)
;
        MVI     C,12    ;NUMBER OF BYTES = 12
PRN1:   DCX     H
        MOV     A,M     ;GET A BYTE FROM STACK
        CALL    PRNT    ;AND PRINT IT
        DCX     H
        DCR     C
        JNZ     PRN1
;
;RESTORE REGISTERS FROM STACK
;
        POP     H       ;RESTORE REGISTERS FROM STACK
        POP     H
        POP     D
        POP     B
        POP     PSW
        RET
```

Note that calling the register dump routine places the old Program Counter in the Stack.

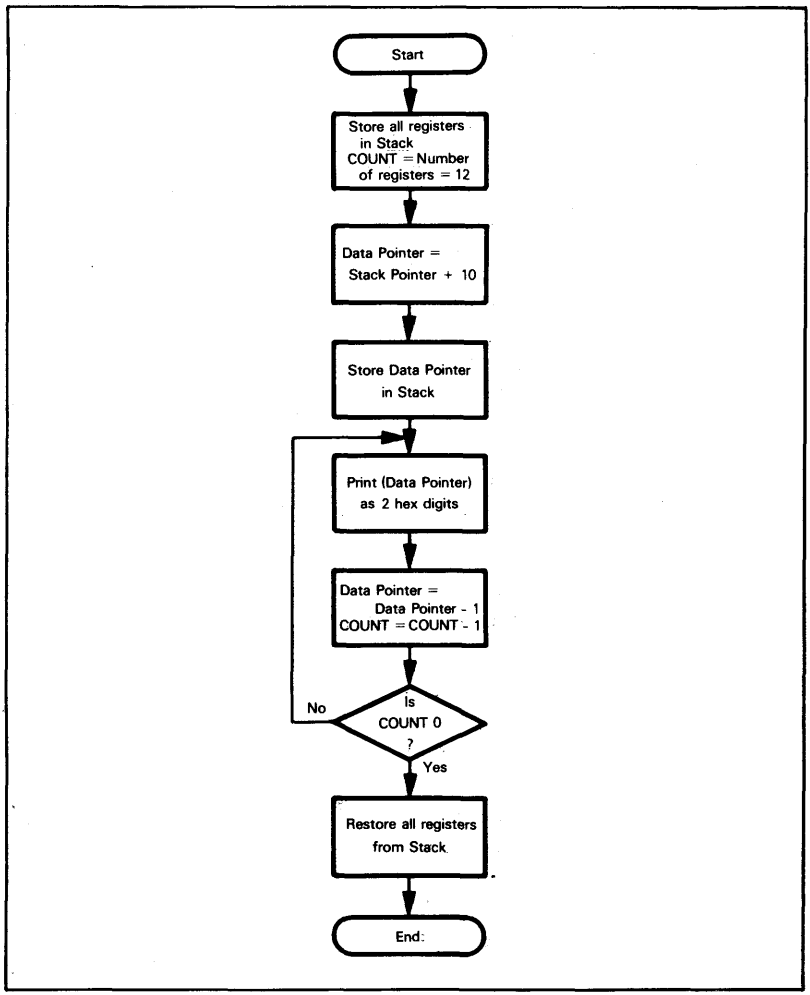

Start
Store all registers in Stack COUNT = Number of registers = 12
Data Pointer = Stack Pointer + 10
Store Data Pointer in Stack
Print (Data Pointer) as 2 hex digits
Data Pointer = Data Pointer - 1 COUNT = COUNT - 1
Is COUNT 0 ?
No
Yes
Restore all registers from Stack
End:

Figure 14-2. Flowchart Of Register Dump Program

```
3E    (PROGRAM COUNTER)
FC
86    (PSW)
1D    (A)
07    (B)
3E    (C)
23    (D)
01    (E)
17    (H)
F0    (L)
FC    (STACK POINTER)
3F
```

Figure 14-3. Results Of A Typical Register Dump

**MEMORY DUMP**

A memory dump is a program which can display the contents of an entire section of memory. This is much more efficient for examining arrays or blocks than single locations. However, very large memory dumps are generally not useful (except to produce scrap paper) because of the sheer mass of information which they produce. They may also take a long time to generate on a slow printer. Small dumps may, however, provide the programmer with a reasonable amount of information which can be observed as a unit. Relationships such as regular repetitions of data patterns or offsets of entire arrays may become obvious. A general dump may be rather difficult to write. The programmer should be careful of the following situations:

1) The size of the memory area exceeds 256 bytes, so a single counter will not suffice.
2) The ending location is an address smaller than the starting location. This should be treated as an error, since the user would seldom want to print the entire memory contents in an unusual order.

```
1000  23  1F  60  54  37  38  3E  00
1008  6E  43  38  17  59  44  98  37
1010  47  36  23  81  E1  FF  FF  5A
1018  34  ED  BC  AF  FE  FF  27  02
```

Figure 14-4. Results Of A Typical Memory Dump

Since the speed of the memory dump is usually dependent on printer speed, which is always much slower than processor speed, the efficiency of the routine seldom matters. The following program will reject cases where the starting address is after the ending address, and will handle blocks of any length. We assume that the starting address is in Registers H and L, the ending address is in Registers D and E:

```
;
;LOOK FOR END BEFORE START
;
        MOV     A,D     ;EXAMINE MSBs OF ADDRESS
        CMP     H
        JC      DONE    ;THROUGH IF START LARGER
        JNZ     DUMP    ;OKAY IF START SMALLER
        MOV     A,E     ;IF MSBs EQUAL, LOOK AT LSBs
        CMP     L
        JC      DONE    ;THROUGH IF START LARGER
;PRINT 1 LOCATION
;
DUMP:   MOV     A,M     ;GET 1 LOCATION
        CALL    PRNT    ;AND PRINT IT
        MOV     A,D     ;EXAMINE MSBs OF ADDRESS
        CMP     H
        JNZ     CONT    ;CONTINUE IF NOT EQUAL TO LIMIT
        MOV     A,E     ;EXAMINE LSBs OF ADDRESS
        CMP     L
        JZ      DONE    ;DONE IF AT LIMIT
CONT:   INX     H
        JMP     DUMP
DONE:   JMP     DONE
```

Figure 14-4 shows the output from a dump of memory location 1000 to 101F. This routine correctly handles the case in which the starting and ending locations are the same. The user will have to interpret the results carefully if the dump area involves the Stack, since the dump subroutine will itself use the Stack. PRNT may also change parts of the memory.

## MORE ADVANCED DEBUGGING TOOLS

The more advanced debugging tools that are most widely used are:

1) Simulator programs to check software.
2) Logic analyzers to check signals and timing.

Many variations of both these tools exist, and we shall only discuss the standard features.

**SIMULATOR**

The simulator is the computerized equivalent of the pencil-and-paper computer. It is a computer program which goes through the operating cycle of another computer program keeping track of the contents of all the registers, flags, and memory locations. We could, of course, do this by hand, but it would require an unacceptably large amount of effort and close attention to the exact effect of each instruction. The simulator program, on the other hand, never gets tired or confused, forgets an instruction or runs out of paper. Most simulators are large FORTRAN programs. You can purchase them or use them on the time-sharing services. The 8080 simulator is available in several versions from different sources.

Typical simulator features are:

1) A breakpoint facility. Breakpoints can usually be set to occur after a particular number of cycles, when a memory location or one of a set of memory locations is referenced, when the contents of a location or one of a set of locations is altered, or on other conditions that are specific to the particular simulator.
2) Register and memory dump facilities which can display the values of memory locations, registers and I/O ports.
3) A trace facility which will print the contents of particular registers or memory locations, whenever the program changes or references them.
4) A load facility which allows the user to set values initially or change them during the simulation.

Some simulators can also simulate input/output, interrupts, and even direct memory access.

The simulator has many advantages:

1) It can provide a complete description of the status of the computer, since the program is not restricted by pin limitations or other problems.
2) It can provide breakpoint, dump, trace and other facilities which do not involve the processor memory. These facilities will therefore not interfere with the user program.
3) Programs, starting points and other conditions are easy to change.
4) All the facilities of large computers, including peripherals and software, are available to the microprocessor designer.

On the other hand, the simulator is limited by its software base and its separation from the real computer. The major limitations are:

1) The simulator usually cannot help with timing problems.
2) The simulator cannot fully re-create the input/output system.
3) The simulator is usually quite slow. Recreating one second of processor time may take hours of computer time. Using the simulator can be quite expensive.

The simulator represents the software side of debugging; it has the typical advantages and limitations of a wholly software approach. Basically, the simulator can provide insight into program logic and other software problems but cannot help with timing, I/O and other hardware problems.

**LOGIC ANALYZER**

The logic or microprocessor analyzer is the hardware solution to debugging. (For more complete information see Runyon, "Focus on Logic and $\mu$P Analyzers" Electronic Design, Feb. 1, 1977, pp. 40-50). Basically, the analyzer is a parallel digital version of the standard oscilloscope. The analyzer displays information in binary or hexadecimal on a video display, and has a variety of triggering events, thresholds, and inputs. Most analyzers also have a memory so that they can display the past contents of the busses.

The standard procedure is to set a triggering event, such as the occurrence of a particular address on the Address Bus or instruction on the Data Bus in a particular address, or the execution of an input or output instruction. One may then look at the sequence of events that preceded or follow the breakpoint. Common problems you can find this way include short noise spikes (or glitches), incorrect signal sequences, overlapping waveforms, and other timing or signal errors. Of course, a software simulator could not provide any hint as to the existence of those errors any more than a logic analyzer could effectively warn the programmer of errors in program logic.

The logic analyzer can also be useful in checking the proper operation of the microcomputer. (See Farnbach, WA., "Bring up Your μP" Electronic Design, July 19, 1976, pp. 80-85).

Logic analyzers vary in many respects. Some of these are:

**IMPORTANT FEATURES OF LOGIC ANALYZERS**

1) Number of input lines. At least 24 are necessary to monitor an 8-bit Data Bus and a 16-bit Address Bus. Still more are necessary for control signals, clocks, and other important inputs.
2) Amount of memory. Each state that is saved will require several words.
3) Maximum frequency. It must be several MHz to handle the fastest processors.
4) Minimum signal pulse width (important for catching glitches).
5) Type and number of triggering events allowed.
6) Methods of connecting to the microcomputer. This may require a rather complex interface.
7) Number of display channels.
8) Binary, hexadecimal or mnemonic displays.
9) Display formats.
10) Signal hold time requirements.
11) Probe capacitance.
12) Single or dual thresholds, adjustable thresholds.

All of these factors are important in comparing logic and microprocessor analyzers, since the entire field is new and unstandardized. A tremendous range of products is already available, and this range will expand in the future.

Logic analyzers, of course, are only necessary for systems with complex timing. Simple applications with low-speed peripherals have few hardware problems which the designer cannot handle with a standard oscilloscope.

## DEBUGGING WITH CHECKLISTS

The designer cannot, of course, possibly check an entire program by hand. But there are certain trouble spots which the designer can check easily. You can use systematic hand-checking to find a large number of errors without resorting to any debugging tools.

**WHAT TO INCLUDE IN CHECKLIST**

The question is where to place the effort. The answer is on points that can be handled with either a yes-no answer or with a simple arithmetic calculation. Do not try to do complex arithmetic, follow all the flags, or try every conceivable case. Limit your hand checking to matters which can be settled easily. Leave the complex matters to the various debugging tools. Insure, however, that you proceed systematically; build your checklist and make sure that the program does the basic processing correctly.

The first thing to do is compare the flowchart or structured program and the actual code. Make sure that every element that appears in one also appears in the other. A simple checklist will do the job. It is easy to simply omit a branch or a processing section.

Next concentrate on the program loops. Make sure that all registers and memory locations used inside the loop have been initialized before they are used. This is a common source of errors; once again a simple checklist will suffice.

Now look at each conditional branch. Select a sample case that should produce a branch and one that should not; try both of them. Is the branch correct or reversed? If the branch involves checking if a number is above or below a threshold, try the equality case. Does it branch the right way? Make sure that your choice is consistent with the problem definition.

Look at the loops as a whole. Try the first and last iterations by hand; these are often troublesome special cases. What happens if the number of iterations is zero, i.e., there is no data or the table has no elements? Does the program work correctly? Programs often will start performing one iteration of the loop before testing for completion of the task or, even worse, decrement counters past zero before checking them.

Check off everything down to the last statement. Do not assume (hopefully) that the first error is the only one in the program. Hand-checking will allow you to get the maximum benefit from debugging, since you will get rid of many simple errors ahead of time.

A quick review of the hand-checking questions:

**HAND-CHECKING QUESTIONS**

1) Is every element of the program design in the program (and vice-versa for documentation purposes)?
2) Are all registers and memory locations used inside loops initialized before they are used?
3) Are all conditional branches correct?
4) Do all loops start and end properly?
5) Are equality cases handled correctly?
6) Are trivial cases handled correctly?

## LOOKING FOR ERRORS

Of course, despite all these precautions (or if you skip over some of them), programs almost never work the first time they are executed. The designer is left with the problem of where to start looking for the mistake. The hand checklist provides a starting place if you did not use it earlier; some of the errors you may not have eliminated are:

**COMMON ERRORS**

1) Failure to initialize variables such as counters, pointers, sums, etc. Do not assume anything is necessarily zero when you start.

2) Inverting the logic of a conditional jump, e.g., using Jump-On-Carry when you meant Jump-On-Not Carry. Remember the effects of a comparison or subtraction:

   (A is the contents of the Accumulator, M the contents of the register or memory location.)

   $$\begin{aligned} \text{ZERO} &= 1 \text{ if } A = M \\ &= 0 \text{ if } A \neq M \\ \text{CARRY} &= 1 \text{ if } A < M \\ &= 0 \text{ if } A \geq M \end{aligned}$$

   Note particularly that CARRY = 0 if A = M, i.e., in the equality case. So, Jump-On-Carry means jump if A < M, Jump-On-Not Carry means jump if A $\geq$ M. If you want the equality case on the other side, try either reversing the role of A and M or adding 1 to M. For example, if you want a jump if A $\geq$ 10, use

   ```
   CPI     10
   JNC     ADDR
   ```

   If, on the other hand, you want a jump if A < 10, use

   ```
   CPI     11
   JNC     ADDR
   ```

3) Updating the counters and pointers in the wrong place or not at all. Be sure there are no paths through a loop which either skip or repeat the updating instructions.

4) Failing to fall through correctly in trivial cases such as no data in a buffer, no tests to be run, or no entries in a transaction. Do not assume that these will never happen unless the program has specifically eliminated them.

Other problems to watch for are:

5) Reversing the order of operands. Remember that the MOV instruction moves the second operand into the first operand. For example, MOV B,A moves A to B, not the other way around.

6) Changing condition flags before you use them.

Remember that INR and DCR affect all the flags except the Carry, while shifts and DAD affect the Carry. Remember also that POP PSW changes all the flags and logical instructions clear the Carry.

7) Failing to change condition flags when you intended to.

The Zero or Sign flags may not represent the current status of the Accumulator, since many instructions (e.g. MOV, LDA, MVI) do not change the flags. Remember also that INX, DCX and CMA affect no flags at all.

8) Confusing values and addresses.

Remember that LXI H,1000H loads HL with the number 1000 (hex) while LHLD 1000H loads HL with the contents of locations 1000 and 1001. A similar distinction applies to LDA COUNT and MVI A,COUNT.

9) Accidentally re-initializing a register or memory location.

Make sure that the jump addresses are placed so that no extra initialization is performed.

10) Confusing numbers and characters.

Remember that the ASCII representations of digits differ from the digits themselves, e.g., ASCII 7 is 37 hex; 07 is the ASCII Bell character.

11) Confusing binary and decimal.

Remember that the BCD representation of a number is different from its binary representation; e.g., BCD 36 is represented by the binary number 00110110. Note that the decimal equivalent of this binary number is 54, not the original BCD 36.

12) Performing subtraction the wrong way around. Be careful also with other operations (like division) that do not commute.

Remember that SUB M and CMP M produce A-M, not M-A.

13) Ignoring the effects of subroutines and macros.

Do not assume that calls to subroutines or references to macros do not change flags, registers or memory locations. Be sure of exactly what effects they have. Note that it is very important to document these effects.

14) Using the Shift instructions improperly.

Remember the precise effects of RAR, RAL, RLC, and RRC. They are all 1-bit circular shifts which affect only the Carry. Note that there is no arithmetic shift (preserving the sign bit) and no right logical shift (ADD A is a left logical shift).

15) Counting the length of an array incorrectly.

Remember that there are five, not four, memory locations included in addresses 100 through 104 inclusive.

16) Forgetting that certain transfers always use the Accumulator.

These are LDA, STA, IN, OUT, LDAX, and STAX. Be especially careful with the last two, which use the specified register pair as an address but use the Accumulator as the source or destination of the data. For example, LDAX B loads the Accumulator with the data from the address in register pair B.

17) Confusing registers and register pairs.

Remember that LXI, LHLD, DCX, INX, DAD, SHLD, POP and PUSH affect pairs of registers, not single registers. On the other hand, MVI, MOV, DCR and INR affect single registers.

18) Forgetting to initialize the Stack.

Remember that you must initialize the Stack Pointer before calling any subroutines or performing any stack operation.

19) Changing a register or memory location before using it.

Remember that MOV, LDA, MVI, etc. change the contents of the destination (but not the source).

**DEBUGGING INTERRUPT-DRIVEN PROGRAMS**

Interrupt-driven programs are particularly difficult to debug since errors may occur randomly. If, for example, the program enables the interrupts a few instructions too early, an error will only occur if an interrupt is received while the program is executing those few instructions. In fact, you can usually assume that randomly occurring errors are caused by the interrupt system. (See Weller, U.J., Assembly Level Programming for Small Computers, Lexington Books, Lexington, Mass., 1975).

Typical errors in interrupt-driven programs are:

1) Forgetting to re-enable the interrupt after accepting one and servicing it.

The processor disables the interrupt system automatically on RESET or on accepting an interrupt. Be sure that no possible sequences fail to enable the interrupt system.

2) Using the Accumulator before saving it; i.e., PUSH PSW must come before any input or output operations.

3) Forgetting to save the Accumulator and flags.

4) Restoring registers in the wrong order.

If the order in which they were saved was

```
PUSH  PSW
PUSH  B
PUSH  D
PUSH  H
```

the order of restoration should be

```
POP   H
POP   D
POP   B
POP   PSW
```

5) Enabling interrupts before establishing all the necessary conditions such as priority, flags, etc.

A checklist can aid here.

6) Leaving results in registers and destroying them in the restoration process.

7) Forgetting that RST leaves an address in the Stack whether you use it or not.

You may have to re-initialize or update the Stack Pointer.

8) Not disabling the interrupt during multi-word transfers or instruction sequences.

Hopefully, these lists will at least give you some ideas as to where to look. Unfortunately, even the most systematic debugging can still leave some truly puzzling problems.

## DEBUGGING EXAMPLES

### Decimal To 7-Segment Conversion

**DEBUGGING A CODE CONVERSION PROGRAM**

The program converts a decimal number in memory location 40 to a 7-segment code in memory location 41. It blanks the display if memory location 40 is not a decimal number.

**Initial Program:** (from flowchart in Figure 14-5)

```
        LDA     40H     ;GET DATA
        CPI     9
        JC      DONE    ;THROUGH IF DATA > 9
        LHLD    SSEG    ;GET ADDRESS OF 7-SEGMENT
                        ;TABLE
        MOV     D,A
        DAD     D       ;FIND ELEMENT BY INDEXING
        MOV     A,M
DONE:   STA     41H     ;GET 7-SEGMENT CODE
HERE:   JMP     HERE
SSEG:   DB      3FH;06H,5BH,4FH,69H
        DB      6DH,70H,07H,7DH,6FH
```

Using the checklist procedure, we were able to find the following errors:

1) The block which cleared RESULT had been omitted.
2) The conditional branch was incorrect.

For example, if the data is 00, CPI 9 sets the Carry since 0 < 9. However, the jump on the opposite condition, i.e. JNC DONE, still did not produce the correct result. Now, the equality case failed since, if the data was 9, CPI 9 cleared the Carry and caused a jump. The correct version is

```
        CPI     10
        JNC     DONE    ;THROUGH IF DATA > 9
```

**Second Program:**

```
        MVI     B,0     ;BLANK DISPLAY
        LDA     40H     ;GET DATA
        CPI     10
        JNC     DONE    ;THROUGH IF DATA > 9
        LHLD    SSEG    ;GET ADDRESS OF 7-SEGMENT TABLE
        MOV     D,A
        DAD     D       ;FIND TABLE ENTRY BY INDEXING
        MOV     A,M     ;GET 7-SEGMENT CODE
DONE:   STA     41H
HERE:   JMP     HERE
SSEG:   DB      3FH,06H,5BH,4FH,69H
        DB      6DH,7DH,07H,7DH,6FH
```

This version was hand-checked successfully. Since the program was simple, the next stage was to single-step through it with real data. The data selected for the trials was:

```
 0          (the smallest number)
 09         (the largest number)
 10         (a border case)
 6B (hex)   (random)
```

The first trial was with zero in location 40 (hex). The program moved along with no apparent errors until it tried to execute the MOV A,M instruction.

The contents of the Address Bus during the data fetch was 0647, an address that did not even exist in the system. Clearly, something had gone wrong.

It was now time for some more hand-checking. Since we knew JNC did the right thing, the error was clearly subsequent to that instruction but before MOV A,M. A hand check showed:

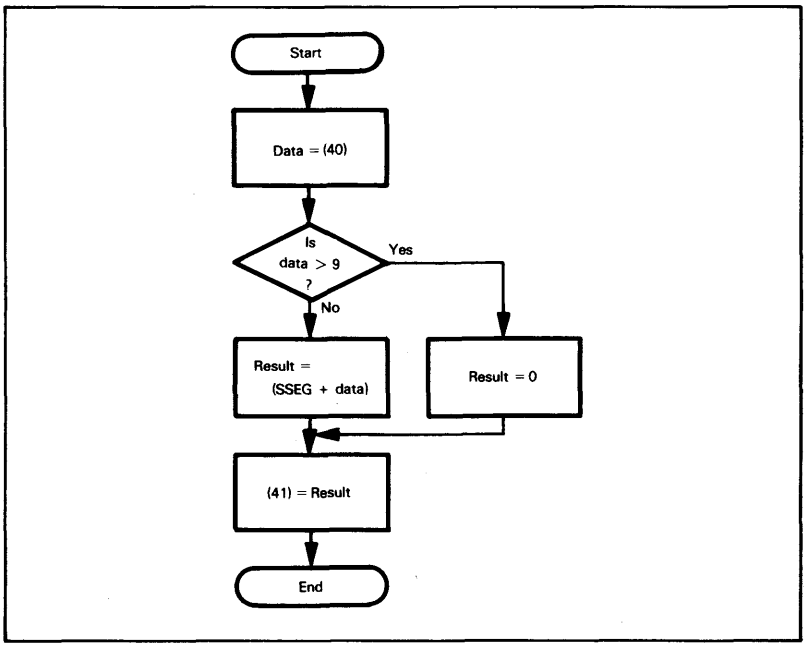



Figure 14-5. Flowchart Of Decimal To 7-Segment Conversion

1) LHLD SSEG places 3F (hex) in L, 06 (hex) in H.

This is clearly wrong. We want LXI, not LHLD, i.e., we want the address SSEG, not the contents of that address.

2) MOV D,A places 0 in D.

This is wrong — the data should go in E, since we want to add it to the least significant bits of the table address. In fact, an instruction should clear D, since the other half of register pair D was not getting initialized at all.

**Third Program:**

```
        MVI     B,0       ;BLANK DISPLAY
        LDA     40H       ;GET DATA
        CPI     10
        JNC     DONE      ;THROUGH IF DATA > 9
        LXI     H,SSEG    ;GET ADDRESS OF 7-SEGMENT TABLE
        MOV     E,A
        MVI     D,0       ;USE DATA AS 16-BIT INDEX
        DAD     D         ;FIND TABLE ENTRY BY INDEXING
        MOV     A,M       ;GET 7-SEGMENT CODE
DONE:   STA     41H
HERE:   JMP     HERE
SSEG:   DB      3FH, 06H, 5BH, 4FH, 69H
        DB      6DG, 7DH, 07H, 7DH, 5FH
```

This program produced the following result:

| DATA | RESULT |
|---|---|
| 0 | 3F |
| 9 | 6F |
| 10 | 10 |
| 6B | 6B |

The program was not clearing the result if the data was invalid, i.e., greater than 10. The program never transferred the blank code in Register B to the Accumulator. Therefore, we altered the program by replacing the MOV A,M instruction with MOV B,M. Thus, at DONE, Register B contains either a) the appropriate 7-segment code if a number between 0 and 9 is present at location 40, or b) 0 if a number other than 0 through 9 is present at location 40. For this reason, a MOV A,B instruction is included at DONE to move the correct data into the Accumulator when the STA 41H instruction is performed. Since the program was simple, it could be tested for all the decimal digits. The results were:

| DATA | RESULT |
|---|---|
| 0 | 3F |
| 1 | 06 |
| 2 | 5B |
| 3 | 4F |
| 4 | 69 |
| 5 | 6D |
| 6 | 7D |
| 7 | 07 |
| 8 | 7D |
| 9 | 6F |

Note that number 8 is wrong — it should be 7F. Since everything else is right, the error is almost surely in the table. In fact, entry 8 in the table had been miscopied.

The final program is:

```
;DECIMAL TO 7-SEGMENT CONVERSION
;
;
        MVI     B,0       ;BLANK DISPLAY
        LDA     40H       ;GET DATA
        CPI     10
        JNC     DONE      ;THROUGH IF DATA > 9
        LXI     H,SSEG    ;GET ADDRESS OF 7-SEGMENT TABLE
        MOV     E,A
        MVI     D,0       ;USE DATA AS 16-BIT INDEX
        DAD     D         ;FIND TABLE ENTRY BY INDEXING
        MOV     B,M       ;GET 7-SEGMENT CODE
DONE:   MOV     A,B
        STA     41H
        HLT
SSEG:   DB      3FH, 06H, 5BH, 4FH, 69H
        DB      6DH, 7DH, 07H, 7FH, 6FH
```

The errors encountered in this program are typical of the ones Intel 8080 assembly language programmers should anticipate. They include:

1) Failing to initialize variables.
2) Inverting conditional branches.
3) Branching incorrectly in the equality case.
4) Confusing immediate and direct addressing, i.e., data and addresses.
5) Forgetting the distinction between 8-bit data and 16-bit addresses.
6) Branching to the wrong place so one path through the program is incorrect.
7) Copying lists of numbers incorrectly.

Note that straightforward instructions like ADD, SUB, JMP, etc. seldom produce any problems.

## Sort Into Decreasing Order

The program sorts an array of unsigned 8-bit binary numbers into decreasing order. The array begins in memory location 41, and its length is in memory location 40.

**DEBUGGING A SORT PROGRAM**

**Initial Program:** (from flowchart in Figure 14-6)

```
        MVI     B,0       ;INTERCHANGE FLAG = 0
        LDA     41H       ;COUNT = LENGTH OF ARRAY
        MOV     C,A
        LXI     H,42H     ;POINT TO START OF ARRAY
PASS:   MOV     A,M       ;GET Kth ELEMENT
        INX     H
        CMP     M         ;COMPARE TO (K+1)th ELEMENT
        JC      CNT       ;NO INTERCHANGE IF Kth LARGER
        MOV     M,A       ;INTERCHANGE IF (K+1)th
                          ;GREATER
        INX     H
CNT:    DCR     C         ;IS PASS COMPLETE?
        JNZ     PASS      ;YES,
        DCR     B         ;IS INTERCHANGE FLAG 0?
        JM      PASS      ;NO, MAKE ANOTHER PASS
                          THROUGH ARRAY
DONE:   JMP     DONE
```

The hand check shows that all the blocks in the flowchart exist in the program and that all the registers have been initialized. The conditional branch must be examined carefully. The internal branch JC CNT must force a branch if the new value is less than or equal to the old value. Note that the equality case must not result in an interchange, since this will create an endless loop with the two equal elements being switched back and forth.

Try an example:

(41) = 30
(42) = 37

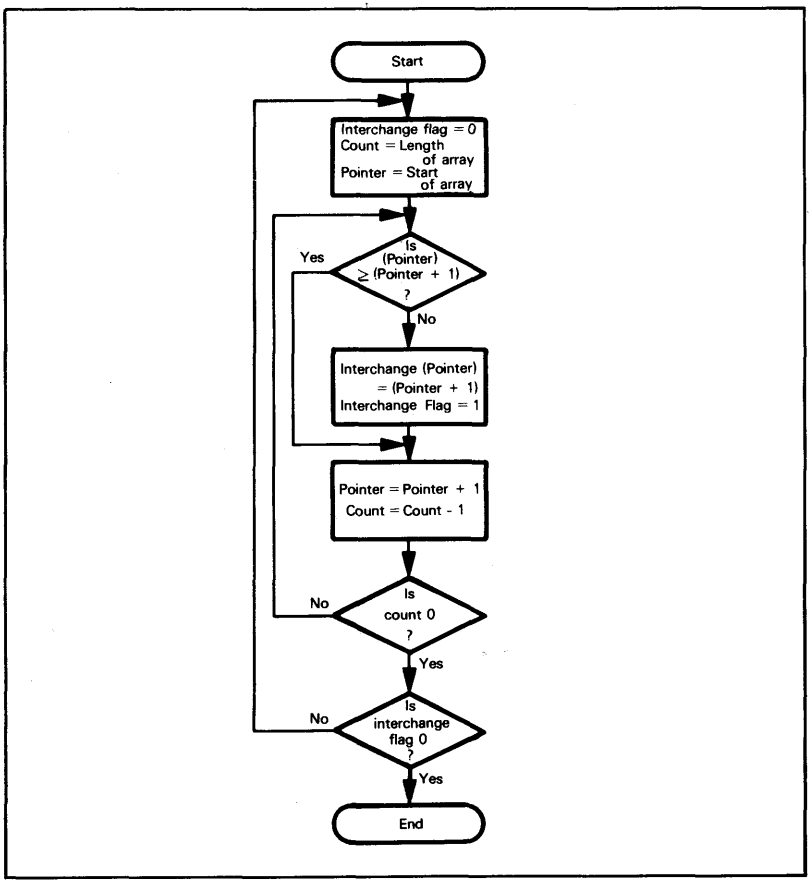



Figure 14-6. Flowchart Of Sort Program

CMP M results in the calculation of 30-37. The Carry is 1. This example should result in an interchange but does not. JNC CNT will provide the proper branch in this case. If the two numbers are equal, the comparison will clear the Carry and JNC CNT is again correct.

How about JM PASS at the end of the program? If there are no interchanges, B will be zero so the branch is wrong. It should be JP PASS.

Now try the first time through. The initialization results in the following:

```
(B)  = 0
(A)  = COUNT
(C)  = COUNT
(HL) = 42
```

The effects of the loop instructions are:

```
MOV   A,M    ;(A)=(42)
INX   H      ;(HL)=43
CMP   M      ;(42)-(43)
JNC   CNT
MOV   M,A    ;(43)=(42)
INX   H      ;(HL)=44
DCR   C      ;(C)=COUNT-1
```

Note that we have already checked the conditional Jump instructions.

Clearly, the logic is incorrect. If the first two numbers are out of order, the results after the first iteration should be:

```
(42) = OLD (43)
(43) = OLD (42)
(HL) = 43
(C)  = COUNT-1
```

Instead, they are:

```
(42) = UNCHANGED
(43) = OLD (42)
(HL) = 44
(C)  = COUNT-1
```

The error in HL is easy to correct. The second INX H is unnecessary and should be omitted. The interchange requires a bit more care and a temporary register, i.e.

```
MOV   D,M
MOV   M,A
DCX   H
MOV   M,D
INX   H
```

An interchange always requires a temporary location where one number can be stored while the other one up is being transferred.

All of these changes require a new copy of the program, i.e.

```
        MVI     B,0       ;INTERCHANGE FLAG = 0
        LDA     41H       ;COUNT = LENGTH OF ARRAY
        MOV     C,A
        LXI     H,42H     ;POINT TO START OF ARRAY
PASS:   MOV     A,M       ;GET Kth ELEMENT
        INX     H
        CMP     M         ;COMPARE TO (K+1)th ELEMENT
        JNC     CNT       ;NO INTERCHANGE IF Kth LARGER
        MOV     D,M       ;INTERCHANGE ELEMENTS
        MOV     M,A
        DCX     H
        MOV     M,D
        INX     H
CNT:    DCR     C         ;IS PASS COMPLETE?
        JNZ     PASS      ;YES
        DCR     B         ;IS INTERCHANGE FLAG 0?
        JP      PASS      ;NO, MAKE ANOTHER PASS THROUGH ARRAY
DONE:   JMP     DONE
```

How about the last iteration? Let's say there are three elements.

(41) = 03
(42) = 02
(43) = 04
(44) = 06

Each time through, the program increments HL by 1. So, at the start of the third iteration,

(HL) = 42+2=44

The effects of the loop instruction are:

```
MOV     A,M       ;(A) = (44)
INX     H         ;(HL) = 45
CMP     M         ;(44) = (43)
```

This is incorrect; the program has tried to move beyond the end of the data. The previous iteration should, in fact, have been the last one, since the number of pairs is one less than the number of elements. The correction is to reduce the number of iterations by 1, i.e., place DCR C after MOV C,A.

How about the trivial cases? What happens if the array contains no elements at all or only one element? The answer is that the program does not work correctly, and without any warning may change a whole block of data improperly (try it!). The corrections for handling the trivial cases are simple but essential; the cost is only a few memory locations to avoid problems that could be very difficult to find later.

The new program is:

```
        MVI     B,0     ;INTERCHANGE FLAG = 0
        LDA     41H     ;COUNT = LENGTH OF ARRAY
        CPI     2       ;IS COUNT 2?
        JC      DONE    ;NO, NO ACTION NECESSARY
        MOV     C,A
        DCR     C       ;NO. OF PAIRS = COUNT-1
        LXI     H,42H   ;POINT TO START OF ARRAY
PASS:   MOV     A,M     ;GET Kth ELEMENT
        INX     H
        CMP     M       ;COMPARE TO (K+1)th ELEMENT
        JNC     CNT     ;NO INTERCHANGE IF Kth LARGER
        MOV     D,M     ;INTERCHANGE ELEMENTS
        MOV     M,A
        DCX     H
        MOV     M,D
        INX     H
CNT:    DCR     C       ;IS PASS COMPLETE?
        JNZ     PASS    ;YES
        DCR     B       ;IS INTERCHANGE FLAG 0?
        JP      PASS    ;NO, MAKE ANOTHER PASS THROUGH ARRAY
DONE:   JMP     DONE
```

Now it's time to check the program on the computer or on the simulator. A simple set of data is:

(41) = 02
(42) = 00
(43) = 01

This set consists of two elements in the wrong order. The program should take two passes. The first pass should rearrange the elements, producing

(42) = 01
(43) = 00
(B) = 01

The second pass should complete the operation and produce

(B) = 00

This program is somewhat long for single-stepping, so we will use breakpoints instead. Each breakpoint will halt the computer and print the contents of all the registers. The breakpoints will come:

1) After LXI H,42H to check the initial conditions.
2) After CMP M to check the comparison.
3) After the second INX H (i.e., just before CNT label) to check the interchange.
4) After DCR B to check the completion of a pass through the array.

The contents of the registers after the first breakpoint were:

| REGISTER | CONTENTS |
|---|---|
| A | 2 |
| B | 0 |
| C | 1 |
| H | 0 |
| L | 42 |

These are all correct, so the program is calculating the initial conditions correctly in this case.

The results at the second breakpoint were:

| REGISTER | CONTENTS |
|---|---|
| A | 0 |
| B | 0 |
| C | 1 |
| H | 0 |
| L | 43 |
| CARRY | 1 |

These results are also correct. The results at the third breakpoint were:

| REGISTER | CONTENTS |
|---|---|
| A | 0 |
| B | 0 |
| C | 1 |
| D | 1 |
| H | 0 |
| L | 43 |

Checking memory showed:

(42) = 01
(43) = 00

The results at the fourth breakpoint were:

| REGISTER | CONTENTS |
|---|---|
| A | 0 |
| B | 0 |
| C | 0 |
| D | 1 |
| H | 0 |
| L | 43 |

Here, Register B is not correct — its value should be 1 to indicate that an interchange occurred. In fact, a look at the program shows that no instruction ever changes B to mark the interchange. The correction is to place the instruction MVI B,1 after JNC CNT.

Now the procedure is to load Register B with the correct value and continue. The second iteration of the second breakpoint gives:

| REGISTER | CONTENTS |
|---|---|
| A | 0 |
| B | 0 |
| C | 0 |
| H | 0 |
| L | 44 |
| CARRY | 1 |

Clearly, the program has proceeded incorrectly without re-initializing the registers (particularly HL). The conditional jump after checking the interchange flag should transfer control all the way back to the start of the program, not to the label PASS.

The final version of the program is:

```
SORT:   MVI     B,0     ;INTERCHANGE FLAG = 0
        LDA     41H     ;COUNT = LENGTH OF ARRAY
        CPI     2       ;IS COUNT ≥ 2?
        JC      DONE    ;NO, NO ACTION NECESSARY
        MOV     C,A
        DCR     C       ;NO OF PAIRS = COUNT-1
        LXI     H,42H   ;POINT TO START OF ARRAY
PASS:   MOV     A,M     ;GET Kth ELEMENT
        INX     H
        CMP     M       ;COMPARE TO (K+1)th ELEMENT
        JNC     CNT     ;NO INTERCHANGE IF Kth LARGER
        MVI     B,1     ;SET INTERCHANGE FLAG
        MOV     D,M     ;INTERCHANGE ELEMENTS
        MOV     M,A
        DCX     H
        MOV     M,D
        INX     H
CNT:    DCR     C       ;IS PASS COMPLETE?
        JNZ     PASS    ;YES
        DCR     B       ;IS INTERCHANGE FLAG 0?
        JP      SORT    ;NO, MAKE ANOTHER PASS THROUGH ARRAY
DONE:   JMP     DONE
```

Clearly, we cannot check all the possible cases for this routine. Two other simple sets of data for debugging purposes are:

1) Two equal elements
   (41) = 02
   (42) = 00
   (43) = 00

2) Two elements already in decreasing order
   (41) = 02
   (42) = 01
   (43) = 00

## INTRODUCTION TO TESTING

Program testing is, clearly, closely related to program debugging. Surely some of the test cases will be the same as the test data used for debugging, i.e.:

**USING TEST CASES FROM DEBUGGING**

1) Trivial cases such as no data or a single element.
2) Special cases which the program singles out for some reason.
3) Simple examples which exercise particular parts of the program.

In the case of the decimal to 7-segment conversion program, these cases cover all the possible situations. The test data consists of

1) The numbers 0 through 9.
2) Boundary case 10
3) The random case 6B

The program does not distinguish any other cases. Here debugging and testing are virtually the same thing.

In the sorting case, the problem is more difficult. The number of elements could range from 0 to 255, and each of the elements could lie anywhere in that range. The number of cases is therefore enormous. Furthermore, the program is moderately complex. How

do we select test data which will in some sense give us a degree of confidence in that program? Here testing requires some design decisions. The testing problem is particularly difficult if the program depends on sequences of real-time data. How do we select the data, generate it, and present it to the microcomputer in a realistic manner?

## TOOLS FOR TESTING

**TESTING AIDS**

Most of the tools mentioned earlier for debugging are helpful in the testing stage also. Logic or microprocessor analyzers can help check the hardware; software simulators can help check the software. Other tools can also be of assistance, e.g.:

1) I/O simulations which can simulate a variety of devices from a single input and a single output device.
2) In-circuit emulators which allow you to attach the prototype and test it from a development system or control panel.
3) ROM simulators which have the flexibility of a RAM but the timing of the particular ROM or PROM that will be used in the final system.
4) Real-time operating systems which can provide inputs or interrupts at specific times (or perhaps randomly) and mark the occurrence of outputs. Real-time breakpoints and traces may also be allowed.
5) Emulations (often on microprogrammable computers) which may provide real-time execution speed and programmable I/O.
6) Interfaces which allow another computer to control the I/O system and test the microcomputer program.
7) Testing programs which check each branch in a program for logical errors.
8) Test generation programs which can generate random data or other distributions.

Formal testing theorems exist, but their applicability is usually limited to very small programs.

You must be careful that the testing equipment does not change the environment so as to invalidate the test. Often testing equipment may buffer, latch, or condition input and output signals. The actual system may not do this, and may therefore behave quite differently.

Furthermore, extra software in the testing environment may use some of the memory space or part of the interrupt system. It may also provide error recovery and other features which will not exist in the final system. A software test bed must be just as realistic as a hardware test bed, since software failure can be just as critical as hardware failure.

Emulations and simulations are, of course, never precise. They are usually adequate for checking logic, but can seldom help test an interface or timing. On the other hand, real-time systems do not provide much of an overview of the logic and may affect the interfacing and timing.

## SELECTING TEST DATA

Very few real programs can be checked in all cases. The designer must choose a sampling that in some sense describes the entire range of possibilities.

**STRUCTURED TESTING**

Testing should, of course, be part of the total development process. Top-down design and structured programming provide for testing as part of the design. This is called "structured" testing (see E. Yourdon, Techniques of Program Structure and Design, Prentice-Hall, Englewood Cliffs, N.J., 1976.), whereby each module within a structured program is separately checked. Testing, as well as programming, should be modular, structured, and top-down.

That leaves the question of selecting test data for a module. The designer must first list all special cases which a program recognizes. These may include:

TESTING SPECIAL CASES

1) Trivial cases
2) Equality cases
3) Special situations

The test data should include all of these.

You must next determine each class of data which statements within the program may distinguish.

These may include:

FORMING CLASSES OF DATA

1) Positive or negative numbers
2) Numbers above or below a particular threshold
3) Data which does or does not include a particular sequence or character
4) Data which is or is not present at a particular time

If each module is small enough, the total number of classes should still be fairly small even though they are multiplicative; i.e., two two-way decisions result in four data classes.

SELECTING TEST DATA FROM CLASSES

You must now distinguish whether the program produces a different result for each entry in the class (as in a table) or produces the same result for each entry (e.g., as in a warning that a parameter is above a threshold). In the discrete case, one may include each element if the number is small or sample if the number of elements is large. The sample should include all boundary cases and at least one case selected randomly. Note that random number tables are available in books and random number generators are part of most computer facilities.

The designer must be careful of distinctions which may not seem obvious. For example, the Intel 8080 will regard an 8-bit unsigned number greater than 127 as negative; the programmer must consider this when using the Jump instructions JM and JP. The programmer must also watch for instructions which do not affect flags, overflow in signed arithmetic, and the distinctions between address length (16-bit) quantities and data length (8-bit) quantities.

## TESTING EXAMPLES

### Sort Program

The special cases here are obvious:

TESTING A SORT PROGRAM

1) No elements in the array
2) One element, the magnitude of which may be selected randomly.

The other special case to be considered is one in which elements are equal.

There may be some problem here with signs and data length. Note that the array itself must be less than 256 elements in length. The use of the instruction MVI B,1 rather than INR B to set the interchange flag means that there will be no difficulty if the number of elements or interchanges exceeds 128.

We could check the effects of the sign by picking half the regular test cases with numbers of elements between 128 and 255 and half between 2 and 127. All magnitudes should be chosen randomly so as to avoid unconscious bias as much as possible.

### Self-Checking Numbers

**TESTING AN ARITHMETIC PROGRAM**

Here we will presume that a prior validity check has ensured that the number is the right length and consists of valid digits. Since the program makes no other distinctions, test data should be selected randomly. Here a random number table or random number generator will prove ideal; the range of the random numbers is 0 to 9.

## TESTING PRECAUTIONS

The designer can simplify the testing procedure by sensible program design. You should use the following rules:

**RULES FOR TESTING**

1) Try to eliminate trivial cases as early as possible without introducing unnecessary distinctions.
2) Minimize the number of special cases. Each special case means additional testing and debugging time.
3) Consider performing validity or error checks on the data prior to processing.
4) Be careful of inadvertent and unnecessary distinctions, particularly in handling signed numbers or using operations that refer to signed numbers.
5) Check boundary cases by hand. These are often a source of errors. Be sure that the problem definition specifies what is to happen in these cases.
6) Make the program as general as reasonably possible. Each distinction and separate routine increases the required testing.
7) Separate and structure the modules so that testing can proceed in steps in conjunction with the other phases of software development.

## CONCLUSIONS

Debugging and testing are the stepchildren of the software development process. Most projects leave far too little time for them and most textbooks neglect them. But designers and managers often find that these stages are the most expensive and time-consuming. Progress may be very difficult to measure or produce. Debugging and testing microprocessor software is particularly difficult because of the limited tools that are available.

The designer should plan debugging and testing carefully. We recommend the following procedure:

1) Try to write programs that can easily be debugged and tested. Modular programming, structured programming, and top-down design are all useful techniques.
2) Prepare a debugging and testing plan as part of the program design. Decide early what data you must generate and what equipment you will need.
3) Debug and test each module as part of the top-down design process.
4) Debug each module's logic systematically. Use checklists, breakpoints, and single-step mode. If the logic is complex, consider the software simulator.
5) Check each module's timing systematically if this is a problem. An oscilloscope can solve many problems if you plan the test properly. If the timing is complex, consider a logic or microprocessor analyzer.
6) Be sure that the test data is a representative sample. Watch for any classes of data which the program may distinguish.
7) If the program handles each element differently or the number of cases is large, select the test data randomly.
8) Record all test results as part of the documentation.

# REFERENCES


1. Collected Algorithms from ACM, ACM Inc., PO Box 12105, Church Street Station, New York 10249
2. Chen, T.C., "Automatic Computation of Exponentials, Logarithms, Ratios, and Square Roots" IBM Journal of Research and Development, Vol. 16, pp. 380-388, July 1972.
3. H. Schmid, Decimal Computation, Wiley-Interscience, New York, 1974
4. Knuth, D.E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1967
5. Knuth, D.E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Addison-Wesley, Reading, Mass., 1969
6. Knuth, D.E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973
7. Carnahan, B., et. al., Applied Numerical Methods, Wiley, New York, 1969
8. Despain, A.M., "Fourier Transform Computers Using CORDIC Iterations", IEEE Transactions on Computers, October 1974, pp. 993-1001.

# Chapter 15
# DOCUMENTATION AND RE-DESIGN

A working program which is driving a computer system does not constitute the end of the software development cycle. Adequate documentation is also an important part of a software product. Not only will documentation help the designer in the testing and debugging stages, it is also essential for later use and extension of the program. A poorly documented program will be difficult to maintain, use, or extend.

Occasionally, the first version of a program does not meet essential requirements of execution time or memory usage. The designer must then consider ways to improve the program. This stage is called re-design, and requires concentration on those parts of the program which can yield the most improvement.

## SELF-DOCUMENTING PROGRAMS

Although no program is ever completely self-documenting, some of the rules we have mentioned earlier can help. These include:

**RULES FOR SELF-DOCUMENTING PROGRAMS**

1) Clear, simple structure with as few transfers of control (i.e., jumps) as possible.
2) Use of meaningful names and labels.
3) Use of names for I/O devices, parameters, numerical factors, etc.
4) Emphasis on simplicity rather than on minor savings in memory usage, execution time, or typing.

For example, the following simple program sends a string of characters to a teletypewriter:

```
        LDA     2000H
        LXI     H,1000H
        MOV     B,A
W:      MOV     A,M
        OUT     6
        CALL    XXX
        DCR     B
        INX     H
        JNZ     W
        HLT
```

Even without adding comments, we can improve the program as follows:

```
MESSG   EQU     1000H
COUNT   EQU     2000H
TTY     EQU     6
        LDA     COUNT
        MOV     B,A
        LXI     H,MESSG
OUTCH:  MOV     A,M
        OUT     TTY
        CALL    BITD
        INX     H
        DCR     B
        JNZ     OUTCH
        HLT
```

Clearly, this program is more comprehensible than the earlier version. Even without adding further documentation, you could probably guess at the function of the program and the meaning of most of the variables. Other documentation techniques are no substitute for self-documentation.

**CHOOSING USEFUL NAMES**

*Some further notes on choosing names:*

1) Use the obvious name when it is available; for example, TTY or CRT for output devices, START or RESET for addresses, DELAY or SORT for subroutines, COUNT or LENGTH for data.
2) Avoid acronyms like S16BA for SORT 16 BIT ARRAY. This kind of label seldom means anything to anyone but the creator.
3) Use full words or close to full words when possible, e.g., DONE, PRINT, SEND.

## COMMENTS

The most obvious form of additional documentation is the comment. However, very few programs (including most of those in books) have really effective comments. You should consider the following guidelines for good comments:

**COMMENTING GUIDELINES**

1) Do not repeat the meaning of the instruction code. Explain what the purpose of the instruction is in the program.

   Comments like:

   ```
           DCR     B       ;B=B-1
   ```

   add nothing to documentation. Instead, include comments like:

   ```
           DCR     B       ;LINE NUMBER=LINE NUMBER-1
   ```

   Remember that you know what the operation codes mean and that any other user can look their meanings up in the manual. The essential purpose of documentation is to explain what the program is doing.

2) Make the comments as clear as possible. Do not use abbreviations or shorthand unless they are really obvious. Avoid comments like:

   ```
           DCR     B       ;LN=LN-1
   ```

   or

   ```
           DCR     B       ;DEC. LN BY 1
   ```

   The extra typing simply is not all that expensive.

3) Comment every important or obscure point. Be particularly careful to mark operations which may not be obvious, such as:

```
ANI     11011111    ;TAPE READER BIT OFF
```

or

```
DAD     D           ;INDEX GRAY CODE TABLE
```

Clearly, I/O operations often require extensive comments. If you are not exactly sure of what an instruction does or you have to think about it, write out the function it performs. The comment will save you time later and will be helpful in documentation.

4) Do not comment the obvious. A comment on each line simply makes it difficult to find the important points. Standard sequences like:

```
INX     H
DCR     B
JNZ     SRCH
```

need not be marked if the purpose of the sequence is obvious due to prior documentation, e.g., if this sequence appeared in a subroutine which added a series of unsigned bytes. One comment will often suffice for several lines. For example:

```
RRC             ;SWAP DIGITS
RRC
RRC
RRC
MOV     A,C     ;EXCHANGE MOST SIGNIFICANT AND LEAST
                ;SIGNIFICANT BYTES
MOV     C,B
MOV     B,A
```

5) Place comments on the line to which they refer or at the start of a sequence.

6) Keep your comments up-to-date. If you change the program, change the comments. Outdated or erroneous comments are worse than none at all.

7) Use standard forms and terms in commenting. Do not worry about repetitiveness; varied names for the same thing are confusing even if it is just COUNT and COUNTER, START and BEGIN, DISPLAY and LEDS, PANEL and SWITCHES. There is no real gain in being inconsistent. The variations may seem obvious to you now but may not be so obvious later; others may be confused from the very beginning.

8) Keep improving your comments. If you come to a comment that you cannot read or understand, take the time to change it. If you find that the listing is getting crowded, add some blank lines. The comments will not improve themselves; in fact, they will just seem to get worse as you leave the task behind and forget exactly what you did.

The bottom line to remember is that comments are important. Good ones will save you time and effort. Put some work into them and try to make them as effective as possible.

## COMMENTING EXAMPLES

### Multiple-Precision Addition



The basic program is:

```
        LDA     40H
        MOV     B,A
        LXI     H,41H
        LXI     D,61H
        ANA     A
ADDW:   LDAX    D
        ADC     M
        MOV     M,A
        INX     D
        INX     H
        DCR     B
        JNZ     ADDW
HERE:   JMP     HERE
```

First, comment the important points. These are typically initializations, data fetches, and processing operations. Do not bother with standard sequences like updating pointers and counters if you have made clear that the registers incremented/decremented are pointers/counters. Remember that names are clearer than numbers, so use them freely.

The new version of the program is:

```
LENG    EQU     40H
NUMB1   EQU     41H
NUMB2   EQU     61H
        LDA     LENG        ;COUNT=LENGTH OF NUMBERS
        MOV     B,A
        LXI     H,NUMB1     ;START AT LSBs OF 1ST NUMBER
        LXI     D,NUMB2     ;START AT LSBs OF 2ND NUMBER
        ANA     A
ADDW:   LDAX    D           ;GET 8 BITS OF 2ND NUMBER
        ADC     M           ;ADD 8 BITS OF 1ST NUMBER
        MOV     M,A         ;STORE RESULT IN 1ST NUMBER
        INX     D
        INX     H
        DCR     B
        JNZ     ADDW
HERE:   JMP     HERE
```

**QUESTIONS FOR COMMENTING**

Second, look for any instructions which might not have an obvious function, and mark them. Here, the purpose of ANA A is to clear the Carry. That purpose certainly is not evident from the instruction.

Third, ask yourself whether the comments tell you what you would need to know if you wanted to use the program, e.g.:

1) What parameters are necessary and what are their starting values?
2) What operations does the program perform?
3) Where does it get the data?
4) Where does it store the results?
5) What special cases does it consider?
6) What does the program do about errors?
7) How does it exit?

Some of the questions may not be relevant to a particular program and some of the answers may be obvious. Just make sure that you will not have to sit down and go through the program to determine the answers. Is there anything you would add to or subtract from this listing? If so, go ahead; you're the one who has to feel that the commenting is adequate and reasonable.

```
LENG    EQU     40H         ;LENGTH OF NUMBERS
NUMB1   EQU     41H         ;LSBs OF 1ST NUMBERS AND RESULT
NUMB2   EQU     61H         ;LSBs OF 2ND NUMBER
        LDA     LENG        ;COUNT=LENGTH OF NUMBERS
        MOV     B,A
        LXI     H,NUMB1     ;START AT LSBs OF 1ST NUMBER
        LXI     D,NUMB2     ;START AT LSBs OF 2ND NUMBER
        ANA     A           ;CARRY= 0 TO BEGIN
ADDW:   LDAX    D           ;GET 8 BITS OF 2ND NUMBER
        ADC     M           ;ADD 8 BITS OF 1ST NUMBER
        MOV     M,A         ;STORE RESULT IN 1ST NUMBER
        INX     D
        INX     H
        DCR     B
        JNZ     ADDW
HERE:   JMP     HERE
```

## Teletypewriter Output

The basic program is:

```
        LDA     40H
        ANA     A
        RAL
        MVI     D,11
TBIT:   OUT     TPORT
        RAR
        STC
        CALL    BITD
        DCR     D
        JNZ     TBIT
HERE:   JMP     HERE
```

Commenting the important points and adding names gives:

```
NBITS   EQU     11          ;NUMBER OF BITS PER CHARACTER
TDATA   EQU     40H         ;CHARACTER TO BE TRANSMITTED
        LDA     TDATA       ;GET A CHARACTER
        ANA     A           ;START BIT=0
        RAL
        MVI     D,NBITS     ;COUNT=NUMBER OF BITS IN CHARACTER
TBIT:   OUT     TPORT       ;BIT TO TTY
        RAR                 ;SERIAL OUTPUT=NEXT BIT
        STC                 ;STOP BIT=1
        CALL    BITD        ;WAIT 1 BIT TIME
        DCR     D
        JNZ     TBIT
HERE:   JMP     HERE
```

Note how simple it would be to change this program so that it would transfer a whole string of data, starting at the address in locations DPTR and DPTR+1, and ending with an "04" character. Furthermore, make the terminal a 30 characters-per-second device with one stop bit (we'll have to change BITD). Try making the changes before looking at the listing.

```
ENDCH   EQU     4           ;ENDING CHARACTER
NBITS   EQU     10          ;NUMBER OF BITS PER CHARACTER
DPTR    EQU     40H         ;(DPTR)=STARTING ADDRESS
        LHLD    DPTR        ;GET STARTING ADDRESS OF MESSAGE
TCHAR:  MOV     A,M         ;GET A CHARACTER
        CPI     NDCH        ;IS IT ENDING CHARACTER?
        JZ      DONE        ;YES DONE
        ANA     A           ;START BIT=0
        RAL
        MVI     D,NBITS     ;COUNT=NUMBER OF BITS IN CHARACTER
TBIT:   OUT     TPORT       ;BIT TO TERMINAL
        RAR                 ;SERIAL OUTPUT=NEXT BIT
        STC                 ;STOP BIT=1
        CALL    BITD        ;WAIT 1 BIT TIME
        DCR     D
        JNZ     TBIT
        INX     H
        JMP     TCHAR
DONE:   JMP     DONE
```

Good comments can make it simple for you to change a program to meet new requirements. For example, try changing the last program so that it:

1) Starts each message with ASCII STX followed by a 3-digit identification code stored in memory locations 30 through 32.
2) Adds no start or stop bits.
3) Waits 1 ms between bits.
4) Transmits 40 characters starting with the one address in the DPTR and DPTR+1.
5) Ends each message with two consecutive ASCII ETX's.

## FLOWCHARTS AS DOCUMENTATION

We have already described the use of flowcharts as a design tool in Chapter 13. Flowcharts also are useful in documentations, particularly if:

**HINTS FOR USING FLOWCHARTS**

1) They are not so detailed as to be unreadable.
2) Decision points are clearly explained and marked.
3) The flowchart includes all branches.
4) The flowchart corresponds to the actual program listing.

Flowcharts are helpful if they give an overall picture of the program. They are not helpful if they are just as difficult to read as an ordinary listing.

## STRUCTURED PROGRAMS AS DOCUMENTATION

Structured programs (see Chapter 13) can also serve as part of the documentation if:

1) You describe the purpose of each section in the comments.
2) You make the range of each conditional or loop structure clear by indentation and ending markers.
3) You make the total structure as simple as possible.
4) You use a consistent, well-defined language.

The structured program can help you to check the logic or improve it. Furthermore, since the structured program is machine-independent, it will ease the implementation of the same task on another computer.

## MEMORY MAPS

A memory map is simply a list of all the memory assignments in a program. The map allows you to determine the amount of memory needed, the locations of data or subroutines, and the parts of memory not allocated. The map is a handy reference for finding storage locations and entry points and for dividing memory between different routines or programmers. The map will also give easy access to data and subroutines if you need them in later extensions or in maintenance.

**TYPICAL MEMORY MAP**

A typical map would be:

Program Memory

| Address | Routine | Purpose |
|---|---|---|
| 0-2 | RESET | TRANSFERS CONTROL TO MAIN PROGRAM IN LOCATION 40 HEX |
| 38-3A | INTRPT | TRANSFERS CONTROL TO INTERRUPT SERVICE IN LOCATION 300 HEX |
| 40-265 | MAIN | MAIN PROGRAM |
| 270-280 | DELAY | DELAY PROGRAM |
| 280-290 | DSPLY | DISPLAY CONTROL PROGRAM |
| 300-340 | KEYIN | INTERRUPT CONTROL PROGRAM FOR KEYBOARD |

Data Memory

| | | |
|---|---|---|
| 1000 | NKEYS | NUMBER OF KEYS |
| 1001-1002 | KPTR | KEYBOARD BUFFER POINTER |
| 1003-1041 | KBFR | KEYBOARD BUFFER |
| 1042-1051 | DBFR | DISPLAY BUFFER |
| 1052-105F | TEMP | TEMPORARY STORAGE |
| 10E0-10FF | STACK | RAM STACK |

The map may also list additional entry points and include a specific description of the unused parts of memory.

## PARAMETER AND DEFINITION LISTS

Parameter and definition lists at the start of the program and each subroutine make understanding and changing the program far simpler. The following rules can help:

**RULES FOR DEFINITION LIST**

1) Separate RAM locations, I/O units, parameters, definition, and memory system constants from each other in the lists.
2) Arrange lists alphabetically when possible, with a description of each entry.
3) Give each parameter that might change a name and include it in the lists. Such parameters may include timing constants corresponding to particular keys or functions, control or masking patterns, starting or ending characters, thresholds, etc.
4) Make the memory system constants into a separate list. These constants will include RESET and interrupt service addresses, the starting address of the program RAM areas, stack areas, etc.
5) Give each separate I/O device a name even though two or more may occupy the same physical port in the current system. The separation will make expansion or reconfiguration much simpler.

A typical list of definitions will be:

**TYPICAL DEFINITION LIST**

```
;
;MEMORY SYSTEM CONSTANTS
;
RESET    EQU   0       ;RESET ADDRESS
INTRP    EQU   38H     ;INTERRUPT ENTRY
START    EQU   40H     ;START OF MAIN PROGRAM
KEYIN    EQU   300H    ;KEYBOARD INTERRUPT PROGRAM
RAMST    EQU   1000H   ;START OF DATA STORAGE
STPTR    EQU   1100H   ;START OF STACK
;
;I/O UNITS
;
DSPLY    EQU   3       ;DISPLAY OUTPUT PORT
KBDIN    EQU   3       ;KEYBOARD INPUT PORT
KBDOT    EQU   3       ;KEYBOARD OUTPUT PORT
TTY      EQU   1       ;TTY DATA PORT
TTYST    EQU   0       ;TTY STATUS PORT
;
;RAM MEMORY
;
         ORG   RAMST
NKEYS    DS    1       ;NUMBER OF KEYS
KPTR     DS    2       ;KEYBOARD BUFFER POINTER
KBFR     DS    40H     ;KEYBOARD INPUT BUFFER
DBFR     DS    10H     ;DISPLAY DATA BUFFER
TEMP     DS    14H     ;TEMPORARY STORAGE
;
;PARAMETERS
;
BOUNCE   EQU   2       ;DEBOUNCING TIME IN MS
GOKEY    EQU   10      ;IDENTIFICATION OF 'GO' KEY
MSCNT    EQU   133     ;COUNT FOR 1 MS DELAY
OPEN     EQU   0FH     ;PATTERN FOR OPEN KEYS
TPULS    EQU   1       ;PULSE LENGTH FOR DISPLAYS IN MS
;
;DEFINITIONS
;
ALL1     EQU   0FFH    ;ALL 1'S PATTERN
STCON    EQU   80H     ;START CONVERSION PUSH
```

Of course, the RAM entries will usually not be in alphabetical order since the designer must order these so as to minimize the number of address changes required in the program.

## LIBRARY ROUTINES

Standard documentation of subroutines will allow you to build up a library of useful programs. The idea is to make these programs as easily accessible as possible. A standard format will allow you or anyone else to see in a glance what the program does. The best procedure is to make up a standard form and use it consistently. Save these programs in a well-organized manner, e.g., according to processor, language, and type of program, and you will soon have a useful set of subroutines. Remember, however, that without organization and proper documentation, using the library may be more difficult than rewriting the program from scratch. Remember that debugging requires a precise understanding of all the effects of each subroutine.

Among the information you will need in a standard form is:

**STANDARD PROGRAM LIBRARY FORMS**

1) Purpose of the program.
2) Processor used.
3) Language used.
4) Parameters required and how they are passed to the subroutine.
5) Results produced and how they are passed to the main program.
6) Number of bytes of memory used.
7) Number of clock cycles required. This number may be an average or typical figure, or it may vary widely. Actual execution time will, of course, depend on the processor clock rate.
8) Registers affected.
9) Flags affected.
10) A typical example.
11) Error handling.
12) Special cases.
13) Documented program listing.

If the program is at all complex, the standard library form should also include a general flowchart or a structured program.

As we have mentioned before, a library program is most likely to be useful if it performs a single distinct function in a reasonably general manner.

## LIBRARY EXAMPLES

### Sum of Data

**Purpose:** The program SUM8 computes the sum of a set of 8-bit unsigned binary numbers.

**Language:** Intel 8080 assembler.

**Initial Conditions:** Starting address of the set of numbers to be added in Registers H and L, length of set in A.

**Final Conditions:** Sum in A.

**Requirements:**

Memory - 9 bytes.
Time - 19+27N clock cycles, where N is the length of the set.
Registers - A,B,H,L.
All flags affected.

**Typical Case:** (all data in hexadecimal):

```
Start:
   (HL)  = 60
   (A)   = 03
   (60)  = 27
   (61)  = 3E
   (62)  = 26
End:
   (A)   = 8B
```

**Error Handling:** Program ignores all carries. Carry bit only reflects the last operation. Initial contents of A must be 1 or more.

**Listing:**

```
;
;SUM OF 8-BIT DATA
;
SUM8:   MOV     B,A     ;COUNT=LENGTH OF DATA BLOCK
        SUB     A       ;SUM=0
ADD8:   ADD     M       ;SUM=SUM+DATA
        INX     H
        DCR     B
        JNZ     ADD8
        RET
```

## Decimal to 7-Segment Conversion

**Purpose:** The program SEVEN converts a decimal number to a 7-segment display code.

**Language:** Intel 8080 assembler.

**Initial Conditions:** Data in A.

**Final Conditions:** 7-segment code in A.

**Requirements:**

Memory - 27 bytes including 7-segment table SSEG.
Time - 78 clock cycles if the data is valid.
Registers - A, B, D, E, H, L.
All flags affected.

**Typical Case:** (data in hexadecimal):

Start:
(A) = 05
End:
(A) = 66

**Error Handling:** Program returns 0FFH if data is not a decimal digit.

**Listing:**

```
;
;DECIMAL TO 7-SEGMENT CONVERSION
;
        MVI     B,0FFH      ;GET ERROR CODE
        CPI     10          ;IS DATA DECIMAL?
        JNC     DONE        ;NO, NO CONVERSION NEEDED
        LXI     D,SSEG      ;GET BASE ADDRESS OF 7-SEGMENT TABLE
        MVI     H,0         ;MAKE DATA INTO 16-BIT INDEX
        MOV     AL
        DAD     D           ;INDEX TABLE
        MOV     B,M         ;GET 7-SEGMENT CODE
DONE:   MOV     A,B
        RET
SSEG:   DB      3FH,06H,5BH,4FH,66H
        DB      6DH,7DH,07H,7FH,6FH
```

## Decimal Sum

**Purpose:** The program DECSM adds two multi-word decimal numbers.

**Language:** Intel 8080 assembler.

**Initial Conditions:** Address of LSBs of one number in H and L, of the other number in D and E. Length of numbers (in words) in A. Numbers arranged starting with LSBs.

**Final Conditions:** Sum replaces number with starting address in H and L.

**Requirements:**

Memory - 13 bytes.
Time - 19+50N where N is the number of words.
Registers - A, B, D, E, H, L.
All flags affected — Carry shows if sum produced a carry.

**Typical Case:** (all data in hexadecimal):

Start:
(HL) = 60
(DE) = 40
(A) = 2
(60) = 34
(61) = 55
(40) = 88
(41) = 15

End:
(60) = 22
(61) = 71
CARRY = 0

**Error Handling:** Program makes no check for validity of decimal inputs. (A) must be 1 or greater. Program does not check for overflow.

**Listing:**

```
;MULTI-WORD
;DECIMAL ADDITION
;
DECSM:  MOV     B,A     ;SAVE LENGTH
        SUB     A       ;CLEAR CARRY TO START
DECAD:  LDAX    D       ;GET 2 DIGITS FROM 2ND NUMBER
        ADC     M       ;ADD 2 DIGITS FROM 1ST NUMBER
        DAA             ;MAKE ADDITION DECIMAL
        MOV     M,A     ;STORE RESULT AS 1ST NUMBER
        INX     D
        INX     H
        DCR     B
        JNZ     DECAD
        RET
```

## TOTAL DOCUMENTATION

Complete documentation of microprocessor software will include all or most of the elements which we have mentioned. The total documentation package may involve:

**DOCUMENTATION PACKAGE**

1) General flowcharts.
2) A written description of the program.
3) A list of all parameters and definitions.
4) A memory map.
5) A documented listing of the program.
6) A description of the test plan and test results.

The documentation may also include:

7) Programmer's flowcharts.
8) Structured programs.

Documentation is not a matter to be taken lightly or to be postponed until the end of the software development cycle. Proper documentation, combined with proper programming practices, is not only an important part of the final product but can also make the development simpler, faster, and more productive. The designer should make consistent and thorough documentation part of every stage of software development.

## RE-DESIGN

Sometimes, the designer may have to squeeze the last microsecond of speed or the last byte of extra memory out of a program. As memories have become larger and less expensive, the memory problem has become less serious. The time problem is, of course, only serious if the application is time-critical; in many applications the microprocessor usually spends most of its time waiting for external devices, and program speed is not a major factor.

Squeezing the last bit of performance out of a program is hardly ever as important as some writers would have you believe. In the first place, the practice is expensive for the following reasons:

**COST OF RE-DESIGN**

1) It requires extra programmer time, which is often the single largest cost in software development.
2) It may sacrifice structure and simplicity with a resulting increase in debugging and testing time.
3) The programs require extra documentation.
4) The resulting programs may be difficult to extend, maintain, or re-use.

In the second place, the lowered cost and performance may not really be justifiable. Will the lowered cost and higher performance really sell more units? Or would you do better with more user-oriented features? The only applications that would seem to justify the extra effort are very high volume, low-cost and low-performance applications, where the cost of an extra memory chip will far outweigh the cost of the extra software development. For other applications, you will find that you are playing an expensive game for no really good reason.

**MAJOR OR MINOR REORGANIZATION**

However, if you must pursue the matter, the following hints will help. First, determine how much more performance or how much less memory you need. If the necessary improvement is 25% or less, you may be able to achieve it by reorganizing the program. If it is more than 25%, either you have made a basic design error, in which case you will need to consider drastic changes in hardware or software, or the designer of the system has placed a ridiculous task in the hands of the software designer. We will first deal with reorganization, and later with drastic changes. You should also look at Chapter 5 of 8080 Programming For Logic Design for some examples.

## REORGANIZING TO USE LESS MEMORY

**SAVING MEMORY**

In general, reorganizing a program to use less memory will also result in higher speed. However, this is not necessarily the case, and higher speed is a less common goal than shorter programs. The methods for achieving shorter programs are:

1) Replace repetitious in-line code with subroutines. Be sure, however, that the CALL and RETURN instructions do not take away most of the gain. Note that this replacement usually results in slower programs because of the time spent in transferring control back and forth.
2) Use register operations when possible. But remember the cost of the extra initialization.
3) Use the stack when possible. The stack is automatically updated each time so that no explicit updating instructions are necessary.
4) Eliminate jump statements. Try to reorganize the program or use indirect jumps (PCHL) or RETURN instructions.
5) Take advantage of addresses which you can manipulate as 8-bit quantities. These include page zero and addresses that are multiples of 100 hexadecimal. For example, you might attempt to concentrate all ROM tables into one $100_{16}$ byte section of memory, and all RAM variables into another $100_{16}$ byte block of memory.
6) Organize data and tables so that you can address them without worrying about carries or without any actual indexing. This will again allow you to manipulate 16-bit addresses as 8-bit quantities. See pages 5-1 to 5-5 of 8080 Programming For Logic Design for an example.
7) Use the 16-bit instructions to replace two separate 8-bit operations. This may be particularly useful in initialization or storing results.
8) Use leftover results from previous sections of the program.
9) Take advantage of such instructions as INR M, DCR M, and MVI M, which operate on memory or place results directly in memory.
10) Use RST instructions to reach subroutines if you are not using them for interrupts.
11) Use INR or DCR to change bit 0 or to move from FF (hex) to zero.

## REORGANIZING TO USE LESS TIME

**SAVING EXECUTION TIME**

Although some of the methods which reduce memory usage will also save time, you can generally only save an appreciable amount of time by concentrating on frequently executed loops. Even completely eliminating an instruction that is only executed once can save at most a few microseconds. But a savings in a loop that is executed frequently will be multiplied many times over.

Therefore, if you must reduce execution time, proceed as follows:

1) Determine which instructions are executed most frequently. You can do this by hand or by using a software simulator or other testing methods.
2) Start with the most frequently executed loops and continue attempting to reduce the number of cycles until you achieve the required reduction.
3) First, see if there are any operations that can be moved outside the loop, i.e., repetitive calculations, data which can be placed in a register or in the stack, addresses which can be placed in registers, special cases or errors which can be handled externally, etc. Note that this will require extra initialization and memory but will save time.
4) Try to eliminate jump statements. These are very time-consuming.
5) Replace subroutines by in-line code. This will save at least a CALL and a RETURN. Note that this method may increase the amount of memory necessary, but a CALL takes 17 cycles and a RETURN takes 10 cycles.
6) Use the Stack for temporary data storage.
7) Use any of the hints mentioned in saving memory which also decrease execution time. These include the use of 8-bit addresses, 16-bit instructions, RST, etc.

## MAJOR REORGANIZATIONS

If you need more than a 25% increase in speed or decrease in memory usage, do not bother to reorganize the code. Your chances of getting that much of an improvement are small unless you call in an outside expert. You are generally better off making a major change.

**BETTER ALGORITHMS**

The most obvious change is a better algorithm. Particularly if you are doing sorts, searches, or mathematical calculations, you may be able to find a faster or shorter method in the literature. Libraries of algorithms are available in some journals and from professional groups. See, for example, references 1 through 8 at the end of the chapter.

More hardware can replace some of the software. Counters, shift registers, hardware multipliers, and other fast add-ons can save both time and memory. Calculators, UARTs, keyboards, encoders, and other slower add-ons may save memory even though they operate slowly.

Other changes may help as well, i.e.:

**OTHER MAJOR CHANGES**

1) A CPU with a longer word length will be faster if the data is long enough. Such a CPU will use less total memory. 16-bit processors, for example, use memory more efficiently than 8-bit processors, since more of their instructions are one word long.
2) Versions of the CPU may exist that operate at higher clock rates. But remember that you will need faster memory and I/O ports and will have to adjust any delay loops.
3) Two CPUs may be able to do the job in parallel or separately if you can divide the job and solve the communications problem.
4) A specially microprogrammed processor may be able to execute the same program much faster. The cost, however, will be much higher even if you buy an off-the-shelf emulation. Such emulations are widely available for the 8080 processor.
5) You can make trade-offs between time and memory. Look-up tables and function ROMs will be much faster than algorithms, but will occupy a lot more memory.

**DECIDING ON A MAJOR CHANGE**

This kind of problem, in which a large improvement is necessary, usually results from lack of adequate planning in the design stage. Part of the problem definition should be to determine which processors and methods will be adequate to handle the problem. If you misjudge, the cost will be high. A cheap solution may result in an unwarranted expenditure of expensive development time. Do not try to just get by; the best solution is usually to do the proper design and chalk a failure off to experience. If you have followed such methods as flowcharting, modular programming, structured programming, top-down design and proper documentation, you will be able to salvage a lot of your effort even if you have to make a major change.

# Chapter 16
# SAMPLE PROJECTS

## PROJECT #1: A Digital Stopwatch

**STOPWATCH INPUT PROCEDURE**

**Purpose:** This project is a digital stopwatch. The operator enters two digits (minutes and tenths of minutes) from a calculator-like keyboard and then presses the GO key. The system counts down the remaining time on two 7-segment LED displays (see Chapter 11 for a description of encoded keyboards and LED displays).

**Hardware:** The project uses one input and one output port, two 7-segment displays, a 12-key keyboard, a 7404 inverter, and a 7408 AND gate. The displays may require drivers, inverters and resistors, depending on their polarity and configuration.

Figure 16-1 shows the organization of the hardware. Output lines 0, 1 and 2 are used to scan the keyboard. Input lines 0, 1, 2 and 3 are used to determine whether any keys have been pressed. Output lines 0, 1, 2 and 3 are used to send BCD digits to the 7-segment decoder/drivers. Output line 4 is used to activate the LED displays (if line 4 is '1', the displays are lit). Output line 5 is used to select the leading or trailing display (output line 5 is '1' for the leading display, '0' for the trailing display). The common line on the leading display is active if line 4 is '1' and line 5 is '1', while the common line on the trailing display is active if line 4 is '1' and line 5 is '0'.

Output line 6 controls the decimal point on the leading display. It may be driven with an inverter or simply left on.

The keyboard is a simple calculator keyboard available for 50¢ from a local source. It consists of 12 unencoded key-switches arranged in four rows of three columns each. Since the wiring of the keyboard does not coincide with the observed rows and columns, the program uses a table to identify the keys. Tables 16-1 and 16-2 contain the input and output connections for the keyboard. The decimal point key is present for operator convenience and for future expansion; the current program does not actually use the key.

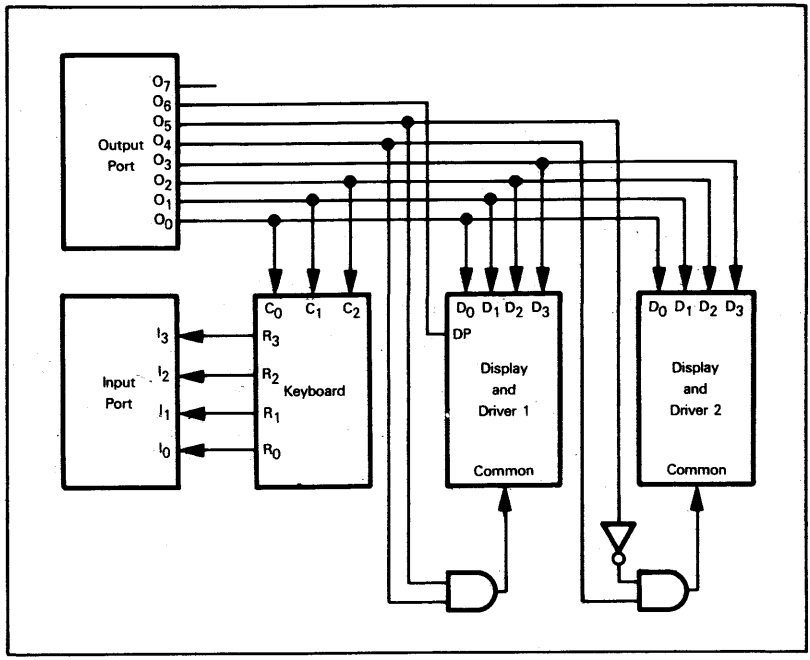



Figure 16-1. I/O Configuration

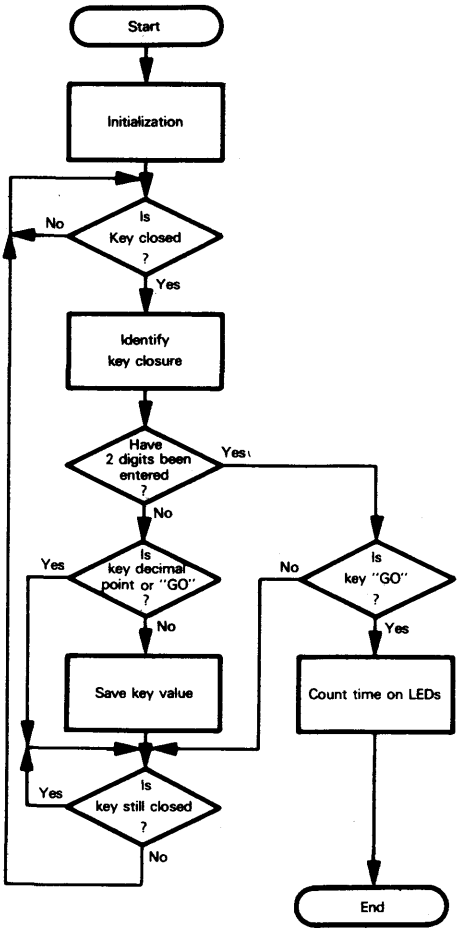
Table 16-1. Input Connections For Timer Keyboard

| Input Bit | Keys Connected |
|---|---|
| 0 | '3', '5', '8' |
| 1 | '2', '6', '9' |
| 2 | '0', '1', '7' |
| 3 | '4', '.', 'GO' |

Table 16-2. Output Connections For Timer Keyboard

| Output Bit | Keys Connected |
|---|---|
| 0 | '0', '2', '3', '4' |
| 1 | '1', '8', '9', 'GO' |
| 2 | '5', '6', '7', '.' |

**General Program Flowchart:**



**Program Description:**

The program is modular and has several subroutines. The emphasis is on clarity and generality rather than efficiency; obviously, the program does not utilize the full capabilities of the 8080 processor. Each section of the listing will now be described in detail.

1) Introductory Comments

   The introductory comments fully describe the program; these comments are a reference so that other users can easily apply, extend and understand the program. Note that standard formats, indentations, and spacings increase the readability of the program.

2) Variable Definitions

   All variable definitions are placed at the start of the program so that they can easily be checked and changed. Each variable is placed in a list alphabetically with other variables of the same type; comments describe the meaning of each variable. The categories are:

   a) Memory system constants, which may vary from system to system depending on the memory space allocated to different programs or types of memories.
   b) Temporary storage (RAM) used for variables.
   c) I/O unit numbers.
   d) Definitions.

   The memory system constants are placed in the definitions so that the user may relocate the program, temporary storage and memory stack without any other changes. The memory constants can be changed to accommodate other programs or to coincide with a particular system's allocation of ROM and RAM addresses.

   Temporary storage is allocated by means of a DS (Define Storage) pseudo-operation. An ORG (Origin) pseudo-operation places the temporary storage locations in a particular part of memory. No values are placed in these locations, so that the program can be burned into a PROM and the system operated from power-up reset without reloading.

   I/O units are always referred to by name so that the unit numbers can easily be changed to handle varied configurations. Each I/O unit is given a separate name, although some units are physically the same in the present configuration.

   The definitions clarify the meaning of certain constants and allow parameters to be changed easily. Definitions are given in the form (e.g., binary, hex, octal, ASCII, decimal) in which their meaning is the clearest. Parameters (such as debounce time) are placed here so that they can be varied with system needs.

3) Initialization

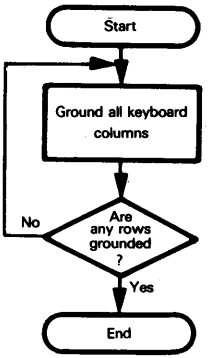
   Location 0 (the reset location on the 8080 microprocessor) contains a jump to the main program. The main program can thus be placed anywhere in memory and reached via a RESET signal.

   The initialization consists of three steps:

   a) Place a starting value in the Stack Pointer. The Stack is only used to store subroutine return addresses.
   b) Start the number of digit keys pressed at zero.
   c) Initialize the location where the next digit key pressed will be saved to the start of the digit key array. Memory location KEYAD contains the address in which the next digit will be placed. Each time the program accepts a digit key, it increments the contents of KEYAD so that it will place the next digit key in the next memory location.

4) Look for Key Closure

**Flowchart:**



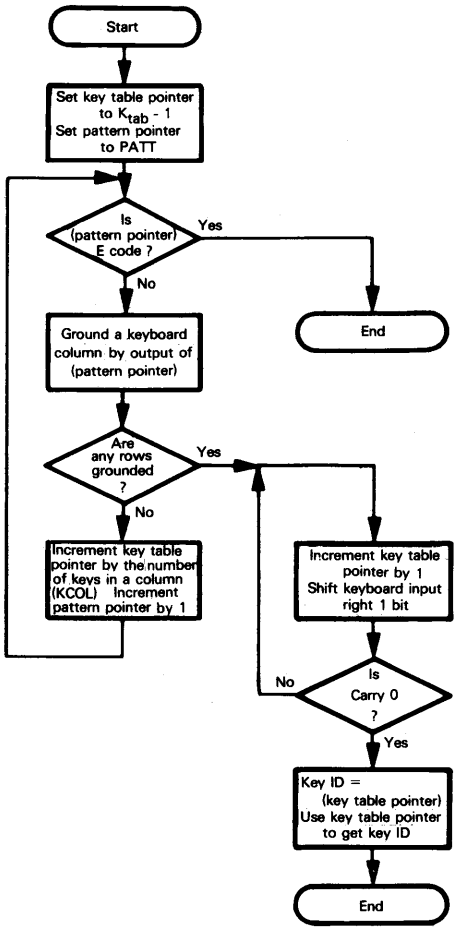

Key closures are identified by grounding all the keyboard columns and then checking for grounded rows (i.e., column-to-row switch closures). Note that the program does not assume any values for the unused input bits; instead, it masks off the bits attached to the keyboard rows with a logical AND instruction.

5) Debounce Key

The program debounces the key closure in software by waiting for two milliseconds, long enough for a clean contact to be made. Subroutine DELAY simply counts with Register C for a millisecond. The number of milliseconds is in the Accumulator. DELAY would have to be adjusted if a slower clock or slower memories were being used. You could make the change simply by redefining the constant MSCNT.

6) Identify Key Closure

**Flowchart:**


Start
Set key table pointer to $K_{tab}$ - 1
Set pattern pointer to PATT
Is (pattern pointer) E code ?
Yes
No
End
Ground a keyboard column by output of (pattern pointer)
Are any rows grounded ?
Yes
No
Increment key table pointer by the number of keys in a column (KCOL) Increment pattern pointer by 1
Increment key table pointer by 1
Shift keyboard input right 1 bit
Is Carry 0 ?
No
Yes
Key ID = (key table pointer)
Use key table pointer to get key ID
End

The particular key closed is identified by grounding single columns and observing whether a closure is found. Once a closure is found (so the key column is known), the key row can be determined by shifting the input.

The patterns required to ground single keyboard columns are in a table PATT in memory. The final pattern in the table is a marker which indicates that all the columns have been grounded without a closure being found. This pattern also indicates to the main program that the closure could not be identified (e.g., the key closure ended or a hardware error occurred before we could find the closure).

**KEY TABLE**

The key identifications are in a table KTAB in memory. The keys in the first column (attached to the least significant output bit) are followed by those in the second column, etc. Within a column, the key in the row attached to the least significant input bit is first, etc. Thus, each time a column is scanned without finding a closure, the number of keys in a column

(KCOL) must be added to the base address of the table to get the starting address of the next column. The key table pointer is also incremented by one before each bit in the row inputs is examined; this process stops when a zero input is found. Note that the key table pointer is started one location before the table, since it is always incremented once in the search for the proper row.

If the program cannot identify the key closure, we simply ignore it and look for another closure.

7) Act on Key Identification

If the program has enough digits (two in this simple case), it only looks for the GO key and ignores all other keys. If it finds the GO key, it proceeds to Step 8 and starts counting.

If the program does not have enough digits, it ignores either the decimal point or the GO key. If it finds a digit key, it saves the value in the key array, increments the number of digit keys pressed, and increments the key array pointer.

If the process is not complete, the program must wait for the key closure to end so that the system will not read the same closure again. The user must wait between key closures, i.e., stop pressing one key before pressing another one. Note that the program will identify double key closures as one key or the other, depending on which closure the identification routine finds first.

8) Set Up Display Output

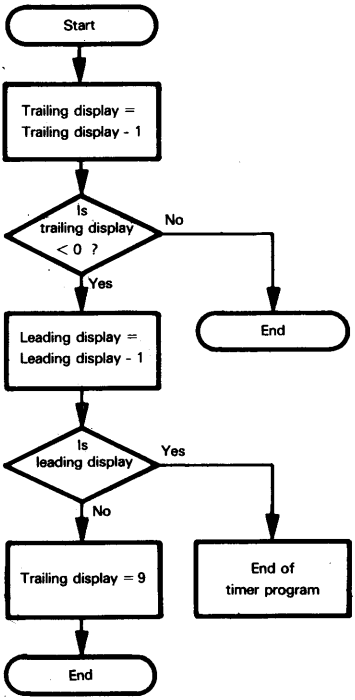
The digits are placed in registers with bit 4 set so that the output is sent to the displays. Bits 5 and 6 are set for the leading display to direct the output correctly and to turn on the decimal point. The logical OR instruction with the appropriate mask sets the bits.

9) Pulse the LED Displays

Each display is turned on for two milliseconds. This process is repeated 1500 times to get a total delay of six seconds. The pulses are frequent enough so that the LED displays appear to be lit continuously.

10) Decrement Display Count

**Flowchart:**




The value on the trailing display is decremented by 1. If this affects bit 4 (LEDON — used to turn the displays on), the value has become negative. A borrow must then be obtained from the leading display. If the borrow from the leading display affects bit 4, the count has gone past zero and the countdown is finished. Otherwise, the program sets the trailing display to '9' and continues.

Note that comments describe both sections of the program and individual statements. The comments explain what the program is doing, not what specific instruction codes do.

```
;PROGRAM NAME: TIMER
;DATE OF PROGRAM: 4/29/76
;PROGRAMMER: LANCE A. LEVENTHAL
;PROGRAM REQUIREMENTS: DC (220) WORDS
;RAM REQUIREMENTS: 5 WORDS
;I/O REQUIREMENTS: 1 INPUT PORT, 1 OUTPUT PORT
;
;
;THIS PROGRAM IS A SOFTWARE TIMER WHICH ACCEPTS INPUTS FROM A
;CALCULATOR-LIKE KEYBOARD AND THEN PROVIDES A STOPWATCH COUNTDOWN
;ON TWO 7-SEGMENT LED DISPLAYS IN MINUTES AND TENTHS OF MINUTES
;
;KEYBOARD
;
;A 12-KEY KEYBOARD IS ASSUMED
;THREE COLUMN CONNECTIONS ARE OUTPUTS FROM THE PROCESSOR
;   SO THAT A COLUMN OF KEYS CAN BE GROUNDED
;FOUR ROW CONNECTIONS ARE INPUTS TO THE PROCESSOR SO THAT
;   COMPLETED CIRCUITS CAN BE IDENTIFIED
;THE KEYBOARD IS DEBOUNCED BY WAITING FOR TWO MILLISECONDS
;   AFTER A KEY CLOSURE IS RECOGNIZED
;A NEW KEY CLOSURE IS IDENTIFIED BY WAITING FOR THE OLD ONE
;   TO END SINCE NO STROBE IS USED
;THE KEYBOARD COLUMNS ARE CONNECTED TO BITS 0 TO 2 OF THE
;   PROCESSOR OUTPUT BUS
;THE KEYBOARD ROWS ARE CONNECTED TO BITS 0 TO 3 OF THE PROCESSOR
;   INPUT BUS
;
;DISPLAYS
;
;TWO 7-SEGMENT LED DISPLAYS ARE USED WITH SEPARATE DECODERS
;   (7447 OR 7448 DEPENDING ON THE TYPE OF DISPLAY)
;THE DECODER DATA INPUTS ARE CONNECTED TO BITS 0 TO 3 OF THE
;   PROCESSOR OUTPUT BUS
;BIT 4 OF THE PROCESSOR OUTPUT BUS IS USED TO ACTIVATE THE LED
;   DISPLAYS (BIT 4 IS '1' TO SEND DATA TO LEDS)
;BIT 5 OF THE PROCESSOR OUTPUT BUS IS USED TO SELECT WHICH LED
;   IS BEING USED (BIT 5 IS '1' IF THE LEADING DISPLAY IS BEING USED,
;   '0' IF THE TRAILING DISPLAY IS BEING USED)
;BIT 6 OF THE PROCESSOR OUTPUT BUS IS USED TO LIGHT THE DECIMAL
;   POINT LED ON THE LEADING DISPLAY (BIT 6 IS '1' IF THE DISPLAY IS
;   TO BE LIT)
;
;METHOD
;
;STEP 1 - INITIALIZATION
;   THE MEMORY STACK (USED FOR SUBROUTINE RETURN ADDRESSES) IS
;   INITIALIZED. THE NUMBER OF DIGIT KEYS PRESSED IS SET TO ZERO,
;   AND THE ADDRESS INTO WHICH THE NEXT DIGIT KEY IDENTIFICATION
;   WILL BE PLACED IS INITIALIZED TO THE FIRST ADDRESS IN THE DIGIT
;   KEY ARRAY
;STEP 2 - LOOK FOR KEY CLOSURE
;   ALL KEYBOARD COLUMNS ARE GROUNDED AND THE KEYBOARD ROWS ARE
;   EXAMINED UNTIL A CLOSED CIRCUIT IS FOUND
;STEP 3 - DEBOUNCE KEY CLOSURE
;   A WAIT OF 2 MS IS INTRODUCED TO ELIMINATE KEY BOUNCE
```

```
;STEP 4 - IDENTIFY KEY CLOSURE
;   THE KEY CLOSURE IS IDENTIFIED BY GROUNDING SINGLE KEYBOARD
;   COLUMNS AND DETERMINING THE ROW AND COLUMN OF THE KEY CLOSURE
;   A TABLE IS USED TO ENCODE THE KEYS ACCORDING TO THEIR ROW AND
;   COLUMN NUMBER
;   IN THE KEY TABLE, THE DIGITS ARE IDENTIFIED BY THEIR VALUES,
;   THE DECIMAL POINT KEY IS NO. 10, AND THE "GO" KEY IS NO. 11
;STEP 5 - SAVE KEY CLOSURE
;   DIGIT KEY CLOSURES ARE SAVED IN THE DIGIT KEY ARRAY UNTIL
;   TWO DIGITS HAVE BEEN IDENTIFIED. DECIMAL POINTS, FURTHER DIGITS,
;   AND CLOSURES OF THE "GO" KEY BEFORE TWO DIGITS HAVE BEEN
;   IDENTIFIED ARE IGNORED
;   AFTER TWO DIGITS HAVE BEEN FOUND, THE "GO" KEY IS USED TO
;   START THE COUNTDOWN PROCESS
;STEP 6 - COUNT DOWN TIMER INTERVAL ON LEDS
;   A COUNTDOWN IS PERFORMED ON THE LEDS WITH THE LEADING DIGIT
;   REPRESENTING THE REMAINING NUMBER OF MINUTES AND THE TRAILING
;   DIGIT REPRESENTING THE REMAINING NUMBER OF TENTHS OF MINUTES
;
;
;
;TIMER VARIABLE DEFINITIONS
;MEMORY SYSTEM CONSTANTS
;
BEGIN   EQU   50H          ;BEGIN IS STARTING MEMORY LOCATION
                           ;FOR PROG
LASTM   EQU   1000H        ;LASTM IS STARTING STACK ADDRESS
TEMP    EQU   800H         ;TEMP IS START OF RAM STORAGE
;RAM TEMPORARY STORAGE
        ORG   TEMP
KEYAD:  DS    2            ;KEYAD HOLDS THE ADDRESS IN THE
                           ;DIGIT KEY ARRAY IN WHICH THE
                           ;IDENTIFICATION OF THE NEXT DIGIT
                           ;KEY WILL BE PLACED
KEYNO:  DS    2            ;KEYNO IS THE DIGIT KEY ARRAY - IT
                           ;HOLDS THE IDENTIFICATIONS OF THE
                           ;DIGIT KEYS WHICH HAVE BEEN PRESSED
NKEYS:  DS    1            ;NKEYS HOLDS NUMBER OF DIGIT
                           ;KEYS PRESSED
;I/O UNITS
;
KBDIN   EQU   3            ;INPUT UNIT FOR KEYBOARD
KBDOT   EQU   3            ;OUTPUT UNIT FOR KEYBOARD
LDOUT   EQU   3            ;OUTPUT UNIT FOR LED DISPLAYS
;
;DEFINITIONS
DECPT   EQU   01000000B    ;CODE TO TURN ON DECIMAL POINT LED
ECODE   EQU   0FFH         ;ERROR CODE FOR NO KEY CLOSURE FOUND
GOKEY   EQU   11           ;IDENTIFICATION NO. FOR "GO" KEY
KCOL    EQU   4            ;NUMBER OF KEYS IN A COLUMN
LEDON   EQU   00010000B    ;CODE TO SEND OUTPUT TO LEDS
LEDSL   EQU   00100000B    ;CODE TO SELECT LEADING DISPLAY
MSCNT   EQU   131          ;COUNT NEEDED TO GIVE 1 MS DELAY TIME
MXKEY   EQU   2            ;MAXIMUM NUMBER OF DIGIT KEY CLOSURES
                           ;USED
```

```
OPEN    EQU     00001111B   ;INPUT FROM KEYBOARD IF NO KEY CLOSED
TPULS   EQU     2           ;NUMBER OF MS BETWEEN DIGIT DISPLAYS
TWAIT   EQU     2           ;NUMBER OF MS TO DEBOUNCE KEYS
;
;
;
        ORG     0
;
;RESET ROUTINE TO REACH TIMER PROGRAM
        JMP     BEGIN       ;FIND TIMER PROGRAM
;INITIALIZATION OF TIMER PROGRAM
;
        ORG     BEGIN
        LXI     SP,LASTM    ;PUT STACK AT END OF MEMORY
        XRA     A
        STA     NKEYS       ;INITIALIZE NO. OF DIGIT KEYS PRESSED
                            ;TO 0
        LXI     H,KEYNO     ;STARTING LOCATION FOR SAVING
                            ;DIGIT KEYS
        SHLD    KEYAD
;
;SCAN KEYBOARD LOOKING FOR KEY CLOSURE
;
START:  CALL    SCANC       ;WAIT FOR KEY CLOSURE
;
;WAIT FOR KEY TO BE DEBOUNCED
;
        MVI     A,TWAIT     ;GET DEBOUNCE TIME IN MS
        CALL    DELAY       ;WAIT FOR KEY TO STOP BOUNCING
;
;IDENTIFY WHICH KEY WAS PRESSED
;
        CALL    IDKEY       ;IDENTIFY KEY CLOSURE
        CPI     ECODE       ;WAS KEY CLOSURE IDENTIFIED BY
                            ;COLUMN SCAN?
        JZ      START       ;IF NOT, WAIT FOR ANOTHER CLOSURE
;ACT ON KEY IDENTIFICATION
        MOV     B,A         ;SAVE KEY NUMBER
        LXI     H,NKEYS
        MOV     A,M         ;CHECK FOR MAXIMUM NO. OF DIGIT
                            ;KEYS
        CPI     MXKEY
        JZ      KEYF        ;IF SO, LOOK FOR "GO" KEY
        MOV     A,B         ;IF NOT, IGNORE DECIMAL POINT OR
                            ;"GO" KEY
        CPI     10
        JNC     WAITK
        INR     M           ;INCREMENT NO. OF DIGIT KEYS PRESSED
        LHLD    KEYAD       ;SAVE IDENTITY OF DIGIT KEY PRESSED
        MOV     M,A
        INX     H           ;MOVE POINTER TO SAVE NEXT DIGIT KEY
        SHLD    KEYAD
;
;WAIT FOR CURRENT KEY CLOSURE TO END
;
```

```
WAITK:  CALL   SCAND        ;WAIT FOR OPEN CIRCUIT
        JMP    START        ;GO LOOK FOR NEXT KEY
;
;LOOK FOR "GO" KEY IF ENOUGH DIGITS FOUND
;
KEYF:   MOV    AB           ;GET NUMBER OF KEY PRESSED
        CPI    GOKEY        ;CHECK FOR "GO" KEY
        JNZ    WAITK        ;IGNORE KEY IF IT'S NOT "GO" KEY
;
;PUT DIGITS INTO REGISTERS FOR DISPLAY
;
        LXI    H,KEYNO
        MOV    A,M          ;GET LEADING DIGIT
        ORI    DECPT        ;TURN ON DECIMAL POINT FOR LEADING
                            ;DIGIT
        ORI    LEDON        ;SET OUTPUT TO LEDS
        ORI    LEDSL        ;SELECT LEADING DISPLAY
        MOV    D,A
        INX    H
        MOV    A,M          ;GET NEXT DIGIT
        ORI    LEDON        ;SET OUTPUT TO LEDS
        MOV    E,A
;
;PULSE THE LED DISPLAYS
;
LEDLP:  MVI    H,6          ;SET COUNTERS FOR 6 SECONDS
TLOOP:  MVI    L,250
LDPUL:  MOV    A,D          ;GET LEADING DIGIT
        OUT    LDOUT        ;OUTPUT TO LED 1
        MVI    A,TPULS      ;DELAY BETWEEN DIGITS
        CALL   DELAY
        MOV    A,E          ;GET TRAILING DIGIT
        OUT    LDOUT        ;OUTPUT TO LED 2
        MVI    A,TPULS      ;DELAY BETWEEN DIGITS
        CALL   DELAY
        DCR    L            ;COUNT DOWN REG L
        JNZ    LDPUL
        DCR    H            ;COUNT DOWN REG H
        JNZ    TLOOP
;
;DECREMENT COUNT ON LED DISPLAYS
;
        DCR    E            ;COUNT DOWN TRAILING DIGIT
        MOV    A,E
        ANI    LEDON
        JNZ    LEDLP        ;OKAY IF NO BORROW NEEDED
        DCR    D            ;IF BORROW NEEDED, COUNT DOWN
                            ;LEADING DIGIT
        MOV    A,D
        ANI    LEDON
        JZ     BEGIN        ;THROUGH IF COUNT PAST ZERO
        MVI    A,9          ;OTHERWISE SET TRAILING DIGIT TO 9
        ORI    LEDON        ;SET OUTPUT TO LEDS
        MOV    E,A
        JMP    LEDLP        ;RETURN TO DISPLAY SECTION
```

```
;
;SUBROUTINE SCANC SCANS THE KEYBOARD WAITING FOR A KEY CLOSURE
;ALL KEYBOARD INPUTS ARE GROUNDED
SCANC:  XRA     A
        OUT     KBDOT           ;GROUND ALL KEYBOARD COLUMNS
        IN      KBDIN
        ANI     OPEN            ;IGNORE UNUSED INPUTS
        CPI     OPEN            ;CHECK FOR CLOSED CIRCUIT
        JZ      SCANC           ;CONTINUE SCANNING IF NO KEYS CLOSED
        RET
;
;
;SUBROUTINE DELAY WAITS FOR THE NUMBER OF MILLISECONDS SPECIFIED
;IN REGISTER A BY COUNTING WITH REGISTER C
DELAY:  MVI     C,MSCNT         ;LOAD REG C FOR 1 MS DELAY
WTLP:   DCR     C               ;WAIT 1 MS
        JNZ     WTLP
        DCR     A               ;COUNT DOWN NUMBER OF MS
        JNZ     DELAY
        RET
;
;
;SUBROUTINE IDKEY DETERMINES THE ROW AND COLUMN NO. OF THE
;KEY CLOSURE AND IDENTIFIES THE KEY ACCORDING TO A TABLE
TDKEY:  LXI     B,PATT          ;POINT TO SCAN PATTERNS TO GROUND
                                ;ONE COL.
        LXI     H,KTAB-1        ;START KEY TABLE POINTER
        LXI     D,KCOL          ;GET NUMBER OF KEYS IN A COLUMN
;
;SCAN KEYBOARD COLUMNS SUCCESSIVELY LOOKING FOR CLOSURE
;
FCOL:   LDAX    B               ;GET PATTERN TO GROUND A PARTICULAR
                                ;COLUMN
        CPI     ECODE           ;HAVE ALL THE COLUMNS BEEN SCANNED?
        RZ                      ;RETURN WITH ERROR CODE IF NO
                                ;CLOSURE FOUND
        OUT     KBDOT           ;SCAN COLUMN
        IN      KBDIN
        ANI     OPEN            ;IGNORE UNUSED INPUTS
        CPI     OPEN            ;CHECK FOR CLOSED CIRCUIT
        JNZ     FROW            ;COLUMN DETERMINED IF CLOSURE
                                ;FOUND
        DAD     D               ;MOVE KEY TABLE POINTER TO NEXT
                                ;COLUMN
        INX     B               ;POINT TO NEXT SCAN PATTERN
        JMP     FCOL
;
;DETERMINE ROW NUMBER OF CLOSURE
;
FROW    TNX     H               ;MOVE KEY TABLE POINTER TO NEXT ROW
        RRC                     ;SHIFT INPUTS LOOKING FOR GROUNDED
                                ;ROW
        JC      FROW            ;KEEP SHIFTING INPUTS UNTIL CLOSURE
                                ;FOUND
```

```
;
;IDENTIFY KEY FROM TABLE
;
        MOV     A,M             ;GET KEY NUMBER
        RET
;SCAN PATTERNS USED TO GROUND A PARTICULAR COLUMN
;ERROR PATTERN USED TO INDICATE THAT ALL COLUMNS HAVE BEEN SCANNED
;THE COLUMN ATTACHED TO OUTPUT BIT 0 IS SCANNED FIRST, THEN
;   THE ONE ATTACHED TO OUTPUT BIT 1, ETC.
PATT:   DB      00000110B,00000101B,00000011B,ECODE
;KEYBOARD TABLE
;COLUMNS ARE PRIMARY INDEX, ROWS SECONDARY INDEX
;THE KEYS IN THE COLUMN ATTACHED TO OUTPUT BIT 0 ARE FOLLOWED
;   BY THOSE IN THE COLUMN ATTACHED TO OUTPUT BIT 1, ETC. WITHIN A
;   COLUMN, THE KEY ATTACHED TO INPUT BIT 0 IS FIRST FOLLOWED
;   BY THE ONE ATTACHED TO INPUT BIT 1, ETC.
;THE DIGIT KEYS ARE 0 TO 9, THE DECIMAL POINT IS 10, AND
;   "GO" IS 11
KTAB:   DB      3,2,0,4,8,9,1,11,5,6,7,10
;
;
;SUBROUTINE SCAND SCANS THE KEYBOARD WAITING FOR KEY CLOSURE
;   TO END SO NEXT CLOSURE CAN BE FOUND
SCAND:  XRA     A
        OUT     KBDOT           ;GROUND ALL KEYBOARD COLUMNS
        IN      KBDIN
        ANI     OPEN            ;IGNORE UNUSED INPUTS
        CPI     OPEN            ;CHECK FOR CLOSED CIRCUIT
        JNZ     SCAND           ;CONTINUE SCAN IF KEY STILL CLOSED
        RET
        END
```

## PROJECT #2: A Digital Thermometer

**Purpose:** This project is a digital thermometer which senses the temperature with a thermistor (through an analog-to-digital converter) and shows the temperature in degrees Celsius on two 7-segment displays.

**Hardware:** The project uses one input and one output port, two 7-segment displays, a 7404 inverter, a 7400 NAND gate or a 7408 AND gate depending on the polarity of the displays, an Analog Devices AD7570J 8-bit monolithic A/D converter, an LM311 comparator, and various peripheral drivers, resistors and capacitors as required by the displays and the converter (see Chapter 11 for a discussion of A/D converters. Also see Hnatek, E.R., A User's Handbook of D/A and A/D Converters, Wiley, New York, 1976; Finkey, J., Computer-Aided Experimentation, Wiley, New York, 1975; Engineering Staff of Analog Devices Inc., Analog-Digital Conversion Handbook, Analog Devices Inc., P.O. Box 796, Norwood, MA, 1972.).

Figure 16-2 shows the organization of the hardware. Output line 7 is used to send a START CONVERSION signal to the A/D converter. Input lines 0 through 7 are attached directly to the eight digital data lines from the converter. Output lines 0 through 3 are used to send BCD digits to the 7-segment decoder/drivers. Output line 4 activates the displays and output line 5 selects the leading or trailing display (line 5 is '1' for the leading display).

**THERMOMETER ANALOG HARDWARE**

The analog part of the hardware is shown in Figure 16-3. The thermistor simply provides a resistance which depends on temperature. Figure 16-4 is a plot of the resistance and Figure 16-5 shows the range of current over which the resistance is constant with current. The conversion to degrees Celsius in the program is performed with a calibration table. The various potentiometers can be adjusted to scale the data properly. A clock for the A/D converter is generated from an RC network as shown in Figure 16-6. The values are R = 33 kΩ, C = 1000 pF so that the clock frequency is about 75 kHz. At this frequency, the maximum conversion time for eight bits is about 100 microseconds. A much longer delay is allowed for conversion so that no check for the end of conversion is necessary.

The 8-bit version of the converter requires the following special connections. The eight data lines are DB2 through DB9 (DB1 is always high during conversion and DB0 low). The Short Cycle 8-bits input (pin 26-SC8; not shown) is tied low so that only an 8-bit conversion is performed. In the present case, High Byte Enable (pin 20-HBEN) and Low Byte Enable (pin 21-LBEN; not shown) were both tied high so that the data outputs were always enabled. A tri-state buffer (the input port shown in Figure 16-1) isolates the outputs from the processor Data Bus.

The converter uses the successive approximation method. It checks each bit with the analog comparator to see if that bit should be on or off. Each comparison takes one clock period. The method is subject to error if the input is noisy or changes during the conversion period.

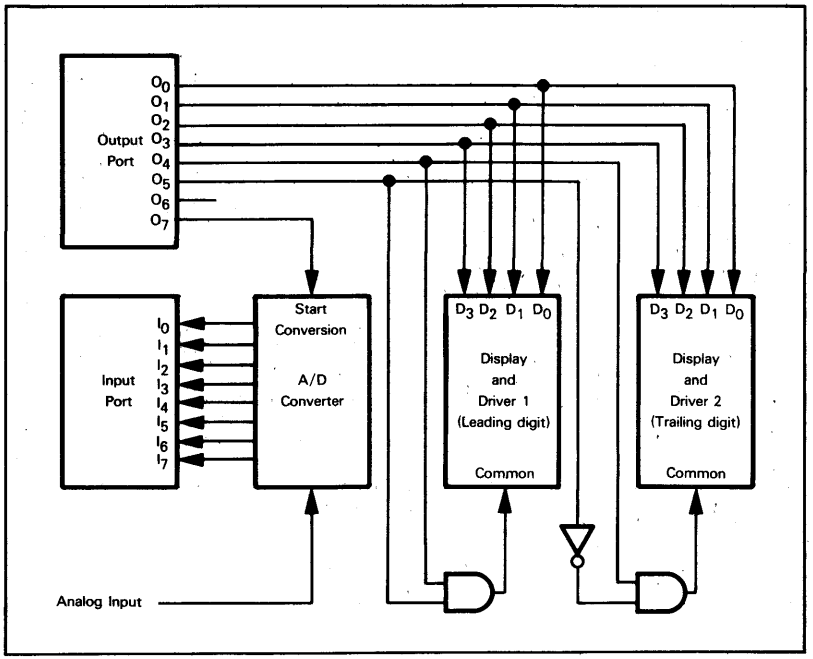
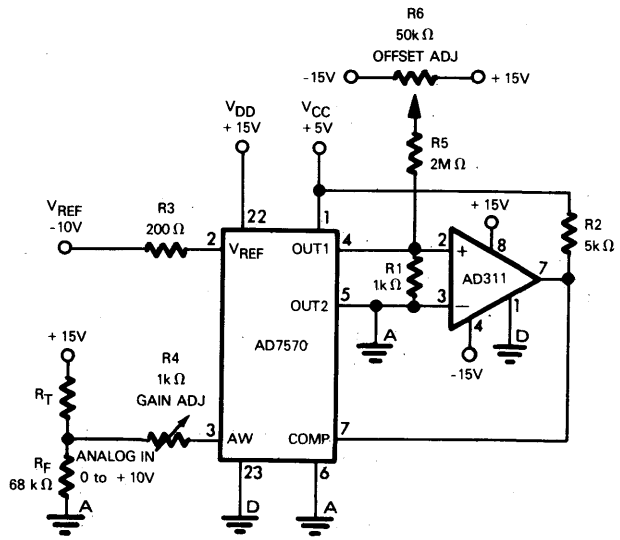



Figure 16-2. I/O Configuration




Note: If positive $V_{REF}$ is used, the ANALOG INPUT range is 0 to $-V_{REF}$, and the COMPARATOR's (-) input should be connected to OUT1 (pin 4) of the AD7570.

$R_T$ is the thermistor. The analog input from the voltage divider is:

$$\frac{R_F}{R_F + R_T} \times 15 \text{ Volt}$$

Since $R_F = 68\ k\Omega$, the input is:

$$\frac{1.02 \cdot 10^6}{R_F + 6.8 \cdot 10^4} \text{ Volt}$$

$R_T$ has a minimum value of $3.4 \cdot 10^4\ \Omega$ (T = 50C, see Figure 16-4) so full scale is 10 Volt.

Figure 16-3. Analog Hardware

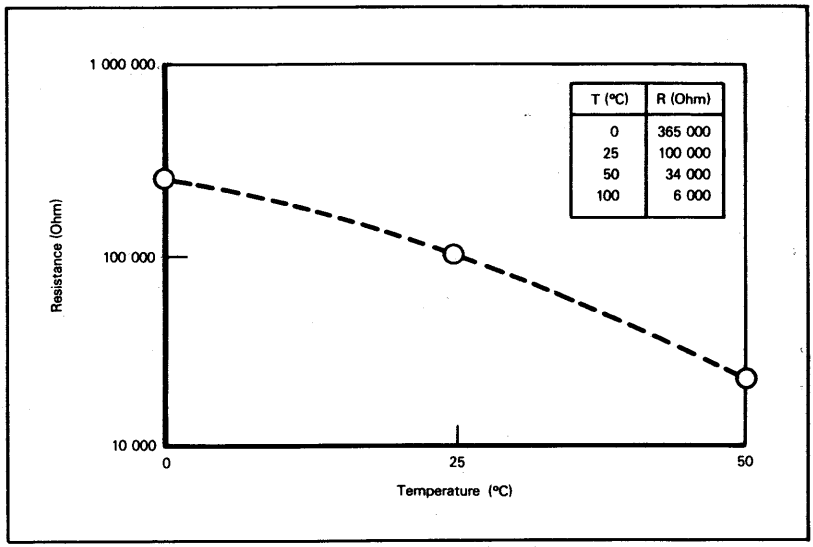



Figure 16-4. Thermistor Characteristics
(Fenwal GA51J1 Bead)

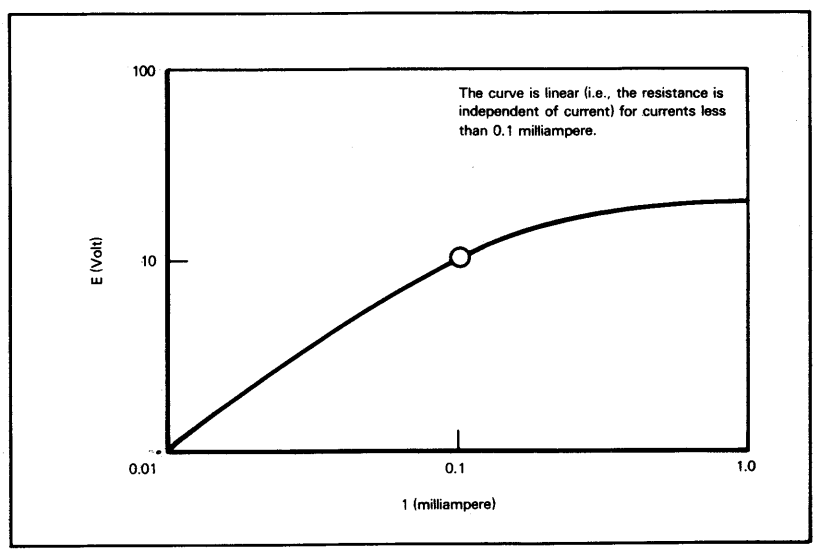



Figure 16-5. Typical E-I Curve for Thermistor (25°C)

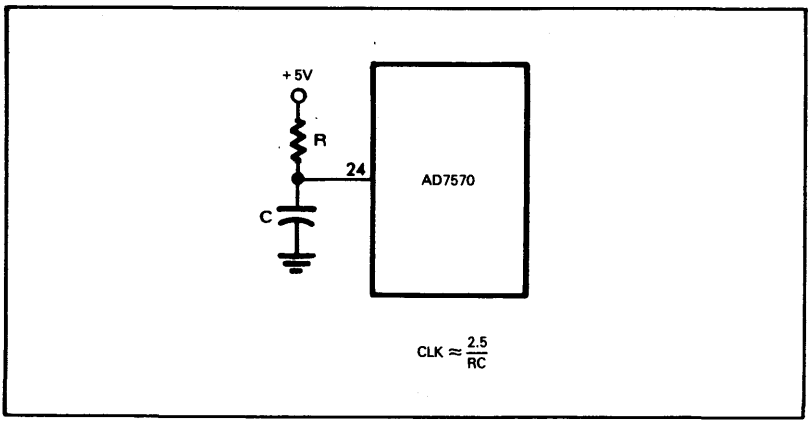
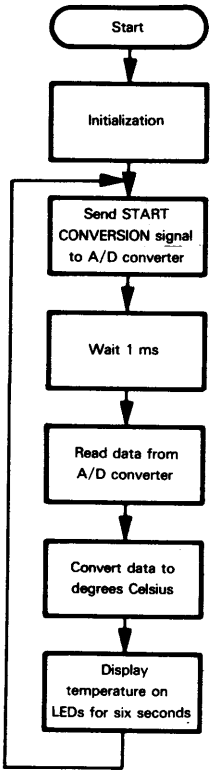



Figure 16-6. Generating an Internal Clock Frequency

**General Program Flowchart:**


Start
Initialization
Send START CONVERSION signal to A/D converter
Wait 1 ms
Read data from A/D converter
Convert data to degrees Celsius
Display temperature on LEDs for six seconds

**Program Description:**

1) Initialization

   Location 0 (the 8080 microprocessor RESET location) contains a jump to the main program. The only initialization is starting the Stack Pointer at the highest address in RAM. The Stack is only used to store subroutine return addresses.

2) Send START CONVERSION signal to A/D converter.

   The CPU pulses the START CONVERSION line by first placing a '1' on output line 7 and then placing a '0' on that line.

   Each input from the converter requires the starting pulse.

3) Wait 1 ms for conversion.

   A delay of 1 ms after the START CONVERSION pulse guarantees a completed conversion. Actually, the converter only takes a maximum of 100 microseconds for an 8-bit conversion. We could reduce the delay by checking the $\overline{\text{BUSY}}$ signal from the converter. This signal will indicate either a '1' (conversion complete) or '0' (conversion in progress) if the BUSY-ENABLE line is addressed with a logic 1. In the present case there is no reason to speed the conversion process.

4) Read data from A/D converter.

   A single input operation reads the data. We should note that the Analog Devices AD7570J has an ENABLE input and tri-state outputs so that it could be tied directly to the microprocessor Data Bus.

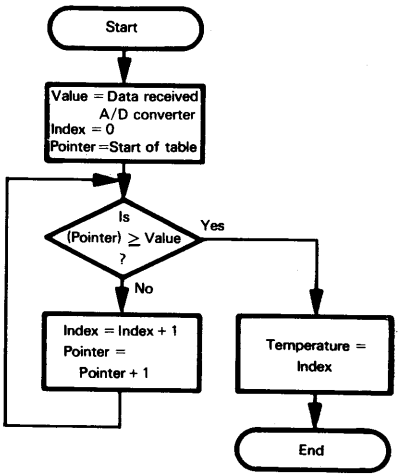
5) Convert data to degrees Celsius.

**USING A CALIBRATION TABLE**

   The conversion uses a table which contains the largest input value corresponding to a given temperature. The program searches the table looking for a value greater than or equal to the value received from the converter. The first such value it finds corresponds to the required temperature; i.e., if the tenth entry is the first value larger than or equal to the data, the temperature is 10 degrees. This search method is inefficient, but the present application does not warrant a better one. (See Knuth, D.E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.) Note that we must keep the entry number in decimal rather than binary. The instruction sequence "ADI 1, DAA" keeps the index as two decimal digits instead of a binary number. For example, the entry number after 9 (00001001 binary) will be decimal 10 (00010000 binary) rather than binary ten (00001010). The reason for this procedure is that we plan to display the temperature as two decimal digits and would have to convert it from binary to decimal otherwise.

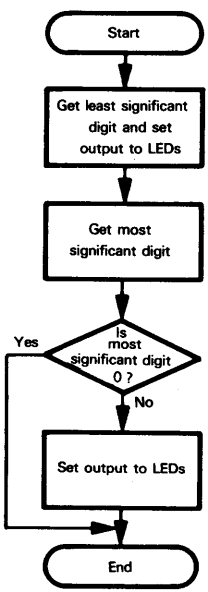
   The table could be obtained by calibration or by a mathematical approximation. The calibration method is the simplest, since the thermometer must be calibrated anyway. The table occupies one memory location for each temperature value.

**Flowchart:**




6) Prepare data for display.

**Flowchart:**



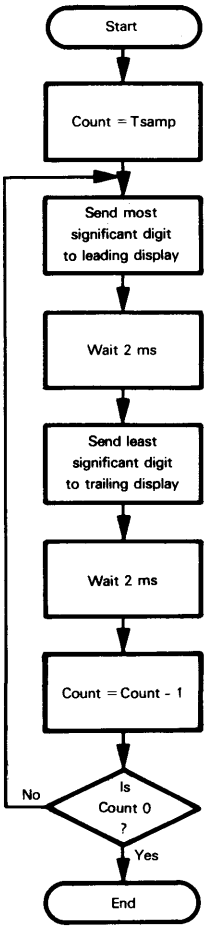
We remove the least significant digit by masking and set the bit which turns on the displays with a logical OR instruction. The result is saved in Register E.

**BLANKING A LEADING ZERO**

The only difference for the most significant digit is that we do not show a leading zero (i.e., the displays show "blank 7" rather than "07" for 7 C). This simply involves not setting the bit which turns on the displays if the digit is zero. The result is saved in Register D.

7) Display temperature for six seconds.

**Flowchart:**



Each display is pulsed often enough so that it appears to be lit continuously. If TPULS were made longer (say 50 ms), the displays would appear to flash on and off.

The program uses a 16-bit counter to count the time between temperature samples. The 8080 has a special instruction —DCX or Double Decrement— to subtract '1' from the 16-bit counter. However, there is no way to directly determine when the counter reaches zero, since DCX does not affect the 8080 status flags. So, we make this determination by logically ORing the eight most significant and the eight least-significant bits of the counter. If that result is zero, the 16-bit counter is zero.

```
;
;PROGRAM NAME: THERMOMETER
;DATE OF PROGRAM: 7/22/76
;PROGRAMMER: LANCE A. LEVENTHAL
;PROGRAM REQUIREMENTS: 159 WORDS
;RAM REQUIREMENTS: NONE
;I/O REQUIREMENTS: 1 INPUT PORT, 1 OUTPUT PORT
;
;
;THIS PROGRAM IS A DIGITAL THERMOMETER WHICH ACCEPTS INPUT FROM
;   AN A/D CONVERTER ATTACHED TO A THERMISTOR, CONVERTS THE INPUT
;   TO DEGREES CELSIUS AND DISPLAYS THE RESULTS ON TWO 7-SEGMENT
;   LED DISPLAYS
;
;A/D CONVERTER
;
;THE A/D CONVERTER IS AN ANALOG DEVICE 7570 MONOLITHIC CONVERTER
;   WHICH PROVIDES AN 8-BIT OUTPUT
;THE CONVERSION PROCESS IS STARTED BY A PULSE ON THE START
;   CONVERSION LINE (OUTPUT BIT 7 OF PORT ADOUT)
;THE CONVERSION IS COMPLETED IN 40 MICROSECONDS AND THE DIGITAL
;   DATA IS LATCHED
;
;
;DISPLAYS
;
;TWO 7-SEGMENT LED DISPLAYS ARE USED WITH SEPARATE DECODERS
;   (7447 OR 7448 DEPENDING ON THE TYPE OF DISPLAY)
;THE DECODER DATA INPUTS ARE CONNECTED TO BITS 0 TO 3 OF THE
;   PROCESSOR OUTPUT BUS
;BIT 4 OF THE PROCESSOR OUTPUT BUS IS USED TO ACTIVATE THE LED
;   DISPLAYS (BIT 4 IS 1 TO SEND DATA TO LEDS)
;BIT 5 OF THE PROCESSOR OUTPUT BUS IS USED TO SELECT WHICH LED
;   IS BEING USED (BIT 5 IS 1 IF THE LEADING DISPLAY IS BEING USED,
;   '0' IF THE TRAILING DISPLAY IS BEING USED)
;
;METHOD
;
;STEP 1 - INITIALIZATION
;   THE MEMORY STACK (USED FOR SUBROUTINE RETURN ADDRESSES) IS IN-
   ITIALIZED
;STEP 2 - PULSE START CONVERSION LINE
;   THE A/D CONVERTER'S START CONVERSION LINE (BIT 7 OF PORT ADOUT)
;   IS PULSED
;STEP 3 - WAIT FOR A/D OUTPUT TO SETTLE
```

```
;   A WAIT OF 1 MS IS INTRODUCED TO ALLOW FOR COMPLETION OF CONVERSION
;STEP 4 - READ A/D VALUE AND CONVERT TO DEGREES C
;   A TABLE IS USED FOR CONVERSION WHICH CONTAINS THE MAXIMUM
;   INPUT VALUE FOR EACH TEMPERATURE READING
;STEP 5 - DISPLAY TEMPERATURE ON LEDS
;   THE TEMPERATURE IS DISPLAYED ON THE LEDS FOR SIX SECONDS BEFORE
;   ANOTHER CONVERSION IS PERFORMED
;
;
;
;THERMOMETER VARIABLE DEFINITIONS
;
;MEMORY SYSTEM CONSTANTS
;
BEGIN   EQU   50H          ;BEGIN IS STARTING LOCATION FOR
                           ;PROGRAM
LASTM   EQU   1000H        ;LASTM IS STARTING ADDRESS FOR
                           ;STACK
;I/O UNITS
;
ADIN    EQU   3            ;INPUT UNIT FOR DATA FROM A/D
                           ;CONVERTER
ADOUT   EQU   3            ;OUTPUT UNIT FOR START CONVERSION
                           ;PULSE TO A/D CONVERTER
LDOUT   EQU                ;OUTPUT UNIT FOR LED DISPLAYS
;
;DEFINITIONS
;
LEDON   EQU   00010000B    ;CODE TO SEND OUTPUT TO LEDS
LEDSL   EQU   00100000B    ;CODE TO SELECT LEADING DISPLAY
MSCNT   EQU   131          ;COUNT USED IN DELAY ROUTINE FOR
                           ;1 MS DELAY
STCON   EQU   10000000B    ;OUTPUT WHICH BRINGS START CONVERSION
                           ;HIGH
STTIM   EQU   1            ;DELAY IN MS TO ALLOW A/D TO DO
                           ;CONVERSION
TPULS   EQU   2            ;DISPLAY PULSE LENGTH IN MS
TSAMP   EQU   1500         ;TSAMP IS THE NUMBER OF TIMES THE
                           ;DISPLAYS ARE PULSED IN A TEMPERATURE
                           ;SAMPLING PERIOD. THE LENGTH OF A
                           ;SAMPLING PERIOD IS THUS 2*TPULS*TSAMP
                           ;MILLISECONDS. THE FACTOR OF
                           ;2*TPULS IS INTRODUCED BY
                           ;THE FACT THAT EACH OF 2
                           ;DISPLAYS IS PULSED FOR TPULS MS
;
;
;
ORG 0
;
;RESET ROUTINE TO REACH THERMOMETER PROGRAM
;
        JMP   BEGIN        ;FIND THERMOMETER PROGRAM
;
;INITIALIZATION OF THERMOMETER PROGRAM
;
```

```
        ORG     BEGIN
        LXI     SP,LASTM        ;PUT STACK AT END OF MEMORY
;
;PULSE START CONVERSION LINE TO GET A/D TO PERFORM A CONVERSION
;
START:  MVI     A,STCON
        OUT     ADOUT           ;START CONVERSION HIGH
        SUB     A
        OUT     ADOUT           ;START CONVERSION LOW
;
;WAIT 1 MS FOR CONVERSION
;
        MVI     A,STTIM         ;CONVERSION DELAY TIME IN MS
        CALL    DELAY           ;WAIT FOR CONVERSION
;
;READ DIGITAL DATA FROM CONVERTER
;
        IN      ADIN            ;GET DATA FROM A/D
;
;CONVERT A/D DATA TO 2 BCD DIGITS
;
        CALL    CONVR           ;CONVERT DATA TO BCD
;
;GET LEAST SIGNIFICANT DIGIT
;
        MOV     B,A             ;SAVE BCD DIGITS
        ANI     0FH             ;MASK OUT LSD
        ORI     LEDON           ;SET OUTPUT TO LEDS
        MOV     E,A             ;SAVE LSD IN REG E
;
;GET MOST SIGNIFICANT DIGIT - BLANK LEADING ZERO
;
        MOV     A,B             ;RESTORE BCD DIGITS
        RRC                     ;SHIFT MSD TO LEAST SIGNIFICANT
                                ;POSITIONS
        RRC
        RRC
        RRC
        ANI     0FH             ;MASK OUT MSD
        JZ      SVMSD           ;DON'T TURN LEADING DISPLAY ON IF
                                ;MSD ZERO
        ORI     LEDON           ;SET OUTPUT TO LEDS
        ORI     LEDSL           ;SELECT LEADING DISPLAY
SVMSD:  MOV     D,A             ;SAVE MSD IN REG D
```

```
;PULSE THE LED DISPLAYS
;
        LXI     H,TSAMP     ;16-BIT COUNTER FOR NO. OF DISPLAY
                            ;PULSES
DSPLY:  MOV     A,D         ;GET LEADING DIGIT
        OUT     LDOUT       ;OUTPUT TO LEADING DISPLAY
        MVI     A,TPULS     ;DELAY DISPLAY PULSE LENGTH
        CALL    DELAY
        MOV     A,E         ;GET TRAILING DIGIT
        OUT     LDOUT       ;OUTPUT TO TRAILING DISPLAY
        MVI     A,TPULS     ;DELAY DISPLAY PULSE LENGTH
        CALL    DELAY
        DCX     H           ;COUNT DOWN 16-BIT COUNTER
        MOV     A,H         ;ARE BOTH 8-BIT SEGMENTS OF
                            ;COUNTER ZERO?
        ORA     L           ;REMEMBER DCX INSTRUCTION DOES
                            ;NOT SET ZERO
        JNZ     DSPLY       ;NO, CONTINUE PULSING DISPLAYS
        JMP     START       ;YES, GO SAMPLE TEMPERATURE AGAIN
;
;SUBROUTINE DELAY WAITS FOR THE NUMBER OF MILLISECONDS SPECIFIED
;IN REGISTER A BY COUNTING WITH REGISTER C
;
;REGISTERS USED: A,C
;
DELAY:  MVI     C,MSCNT     ;LOAD REG C FOR 1 MS DELAY
WTLP:   DCR     C           ;WAIT 1 MS
        JNZ     WTLP
        DCR     A           ;COUNT DOWN NUMBER OF MS
        JNZ     DELAY
        RET
;
;SUBROUTINE CONVR CONVERTS INPUT FROM A/D CONVERTER TO DEGREES
;CELSIUS BY USING A TABLE. INPUT DATA IS IN THE ACCUMULATOR
;AND RESULT IS 2 BCD DIGITS IN ACCUMULATOR
;
;REGISTERS USED: A,B,C,H,L
;
CONVR:  LXI     H,DEGTB     ;GET BASE ADDRESS OF CONVERSION
                            ;TABLE
        MOV     B,A         ;SAVE A/D INPUT
        MVI     C,0         ;START DEGREES AT ZERO
CHVAL:  MOV     A,M         ;GET ENTRY FROM TABLE
        CMP     B           ;IS A/D INPUT LESS THAN OR
                            ;EQUAL TO ENTRY?
        MOV     A,C         ;GET DEGREES CELSIUS
        JNC     FOUND       ;YES, RIGHT VALUE FOUND
        ADI     1           ;ADD 1 TO DEGREES
        DAA                 ;MAKE DEGREES BCD
        MOV     C,A
        INX     H           ;NEXT TABLE ENTRY
        JMP     CHVAL
FOUND:  RET                 ;RETURN WITH TEMP AS 2 BCD
                            ;DIGITS IN A
```

```
;
;
;TABLE DEGTB WAS OBTAINED BY CALIBRATION WITH A KNOWN REFERENCE
;DEGTB CONTAINS THE LARGEST INPUT VALUE WHICH CORRESPONDS TO A
;   PARTICULAR TEMPERATURE READING (I.E., THE FIRST ENTRY IS DECIMAL 58
;   SO AN INPUT VALUE OF 58 IS THE LARGEST VALUE GIVING A ZERO TEM-
PERATURE
;   READING - VALUES BELOW ZERO ARE NOT ALLOWED)
;
DEGTB:  DB      58,61,63,66,69,71,74,77,80,84,87,90,93,97,101,104,108
        DB      112,116,120,124,128,132,136,141,145,149,154,158,
                163,167
        DB      172,177,181,186,191,195,200,204,209,214,218,223,
                227,232
        DB      236,241,245,249,253,255
        END
```

LANCE A. LEVENTHAL is an independent consultant specializing in microprocessors and microprogramming, and is also an instructor in the Engineering and Technology Department at Grossmont College in San Diego, California. He serves as technical editor of the Society for Computer Simulation and as editor of "Microprocessors in Simulation" for Simulation magazine. He is a national lecturer on microprocessors for the IEEE, the author of over thirty articles on microprocessors, and a regular contributor to such publications as Simulation, Electronic Design and Kilobaud.

Dr. Leventhal's previous experience includes affiliations with Linkabit Corporation, Intelcom Rad Tech, Naval Electronics Laboratory Center and Harry Diamond Laboratories. He received a B.A. degree from Washington University in St. Louis, Missouri, and M.S. and Ph.D. degrees from the University of California at San Diego. He is a member of SCS, ACM, and IEEE.

## OTHER BOOKS BY OSBORNE & ASSOCIATES, INC.

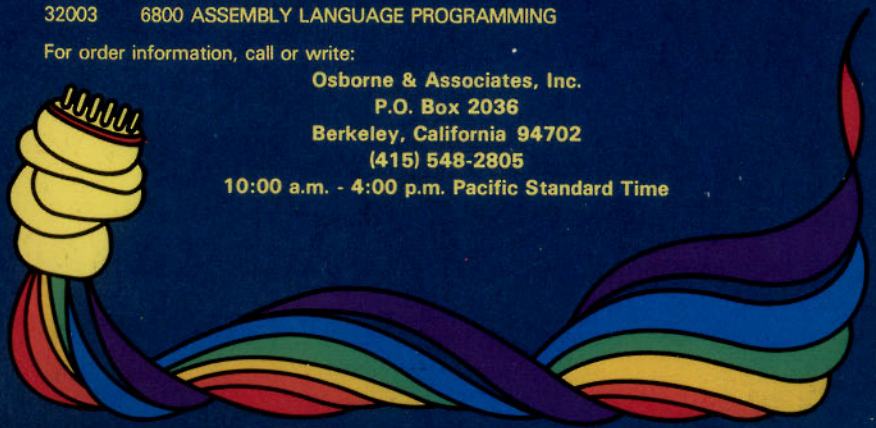
| | |
|---|---|
| 6001 | AN INTRODUCTION TO MICROCOMPUTERS: VOLUME 0 — THE BEGINNER'S BOOK |
| 2001 | AN INTRODUCTION TO MICROCOMPUTERS: VOLUME I — BASIC CONCEPTS |
| 3001 | AN INTRODUCTION TO MICROCOMPUTERS: VOLUME II — SOME REAL PRODUCTS |
| 4001 | 8080 PROGRAMMING FOR LOGIC DESIGN |
| 5001 | 6800 PROGRAMMING FOR LOGIC DESIGN |
| 7001 | Z80 PROGRAMMING FOR LOGIC DESIGN |
| 21002 | SOME COMMON BASIC PROGRAMS |
| 22002 | PAYROLL WITH COST ACCOUNTING |

To be published in 1978:

| | |
|---|---|
| 23002 | ACCOUNTS PAYABLE AND ACCOUNTS RECEIVABLE |
| 24002 | GENERAL LEDGER |
| 32003 | 6800 ASSEMBLY LANGUAGE PROGRAMMING |

For order information, call or write:

**Osborne & Associates, Inc.**
**P.O. Box 2036**
**Berkeley, California 94702**
**(415) 548-2805**
**10:00 a.m. - 4:00 p.m. Pacific Standard Time**



8080A/8085 ASSEMBLY LANGUAGE PROGRAMMING

31003